RTB | RWANDA TVET BOARD

RQF LEVEL 4

GENCP401

COMPUTER SYSTEM AND ARCHITECTURE

C Programming

TRAINEE'S MANUAL

October, 2024

C PROGRAMMING

KOICA Korea International Cooperation Agency

TQUM TVET Quality Management Project

2024

AUTHOR'S NOTE PAGE (COPYRIGHT)

Copies available from:

- *HQs: Rwanda TVET Board-RTB*
- *Web: www.rtb.gov.rw*
- **KIGALI-RWANDA**

Original published version: October 2024

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer's and trainee's manuals for the TVET Certificate IV in Computer system and Architecture, specifically for the module "GENCP401 C Programming".

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda.

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

This training manual was developed:

Under Rwanda TVET Board (RTB) guiding policies and directives

Under Financial and Technical support of

COORDINATION TEAM

RWAMASIRABO Aimable
MARIA Bernadette M. Ramos
MUTIJIMA Asher Emmanuel

Production Team

Authoring and Review

UWAMAHORO Bonaventure

Validation

MUNYANEZA Alphonse

NIYONSHUTI Yves

Conception, Adaptation and Editorial works

HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

KWIZERA INGABIRE Diane

Formatting, Graphics, Illustrations, and infographics

YEONWOO Choe
SUA Lim
SAEM Lee
SOYEON Kim
WONYEONG Jeong
HAKIZAYEZU Adrien

Financial and Technical support

KOICA through TQUM Project

TABLE OF CONTENT

ACRONYMS

ALGOL: Algorithmic Language

ALU: Arithmetic Logic Unit

API: Application Programming Interface

APL: A Programming Language

ASCII: American Standard Code for Information Interchange

BASIC: Beginners All Purpose Symbolic Instruction Code

COBOL: Common Business Oriented Language

CSS: Cascading Style Sheets

DLL: Dynamic Link Library

DMA: Direct Memory Access

EOF: End of File

FORTRAN: Formula Translation

GCC: GNU Compiler Collection

GPU: Graphics Processing Unit

GUI: Graphical User Interface

IDE: Integrated Development Environment

IO: Input/output

IOT: Internet of things

JIT: Just-In-Time compilation

JSON: JavaScript Object Notation

MVC: Model-View-Controller

OOP: Object-Oriented Programming

OS: Operating System

PL/I: Programming Language, Version 1

Prolong: Program in Logic

PSU: Power Supply Unit

RPG: Report Program Generator

INTRODUCTION

This trainee's manual includes all the knowledge and skills required in Computer system architecture specifically for the module of **"C Programming".** Trainees enrolled in this module will engage in practical activities designed to develop and enhance their competencies. The development of this training manual followed the Competency-Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

The trainee's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

As a trainee, you will start by addressing questions related to the activities, which are designed to foster critical thinking and guide you towards practical applications in the labor market. The manual also provides essential information, including learning hours, required materials, and key tasks to complete throughout the learning process.

All activities included in this training manual are designed to facilitate both individual and group work. After completing the activities, you will conduct a formative assessment, referred to as the end learning outcome assessment. Ensure that you thoroughly review the key readings and the 'Points to Remember' section.

MODULE CODE AND TITLE: GENCP401 C PROGRAMMING

Learning Outcome 1: Apply Computer programming Languages
Learning Outcome 2: Write C Programming Codes
Learning Outcome 3: Perform Program Testing


C++
Ruby
Perl
C
C++
Java
Delphi
Pascal
Program
C++
Python
Python

Indicative contents
1.1 Identification of programming Language **1.2 Development of an algorithm** **1.3 Development of a flowchart**

Key Competencies for Learning Outcome 1: Describe Computer Programming Languages

Knowledge	Skills	Attitudes
• Description of programming Languages. • Identification of programming IDE. • Description of flowchart . • Description of algorithm.	• Installing c programming IDE • Analysing a given problem • Arranging steps(logical sequence) according to the given problem • Designing flowchart • Developing an algorithm	• Being Patient • Have Critical thinking, and Attentive • Having Team Work and creativity • Having Team Work, and spirit

Duration: 10 hrs

Learning outcome 1 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly different types of programming languages as used in computer programming
2. Install properly C programming IDE according to the computer specifications.
3. Describe correctly different flowchart symbols according to their functions.
4. Design properly the flowchart according to the problem to be solved.
5. Develop properly the algorithms according to the problem to be solved.

Resources

Equipment	Tools	Materials
• Computers	• C programming ID	• Internet

Indicative content 1.1: Identification of programming language

Duration: 5 hrs

Theoretical Activity 1.1.1: Identification of different types of computers programming

Tasks:

1: In small groups, you are requested to answer the following questions related to the Identification of different types of computer programming languages.

i. What is a computer programming language?
ii. What is a computer programming IDE?
iii. Enumerate 2 role of programming Computer system architecture
iv. Give 2 types of programming languages.

2: Provide the answer for the asked questions and write them on papers.
3: Present the findings/answers to the whole class
4: Follow the expert view from the trainers
5: Ask questions for more clarification where necessary.
6: Read the key **readings 1.1.1.**

Key readings 1.1.1.: Identification of different types of computers programming

1. Types of programming Languages

A computer programming language is a formal language comprising a set of instructions that produce various kinds of output. These languages are used to implement algorithms and to express computations in a format that can be understood and executed by computers.

- **Low-Level Languages**
  Low-level languages are programming languages that provide little or no abstraction from a computer's instruction set architecture. These languages are closely related to the hardware and are used to write programs that interact directly with the hardware.

- ✓ **Machine Language**: The most basic type of programming language, consisting of binary code (0s and 1s) that is directly understood by the computer's hardware.
- ✓ **Assembly Language**: A step above machine language, it uses symbolic names (mnemonics) to represent binary code. Requires an assembler to convert into machine language.

- **High-Level Languages**

  High-level programming languages are languages that are designed to be easy for humans to read, write, and understand. They provide a high degree of abstraction from the machine's hardware, allowing programmers to write code in a more natural and logical way, using words and symbols closer to human language.

  ✓ **Procedural Languages**: Focus on procedure or routine. Examples include C, Pascal, and Fortran.

  ✓ **Object-Oriented Languages**: Centered around objects and classes. Examples include C++, Java, and Python.

  ✓ **Functional Languages**: Based on mathematical functions. Examples include Haskell and Lisp.

  ✓ **Scripting Languages**: Used for automating tasks. Examples include Python, JavaScript, and Ruby.

  ✓ **Logic Programming Languages**: Based on formal logic. Example includes Prolog.

  ✓ **Markup Languages**

  Used for annotating a document in a way that is syntactically distinguishable from the text. Examples include HTML, XML, and LaTeX.

  ✓ **Domain-Specific Languages**

  Tailored to specific application domains. Examples include SQL for databases and MATLAB for mathematical computations.

  2. Role of Programming Languages in Computer system and architecture

  ✓ **Embedded Systems**: Programming languages like C, C++, and Assembly are used to develop firmware and software for microcontrollers and processors in embedded systems.

  ✓ **Signal Processing**: Languages like MATLAB and Python (with libraries like NumPy and SciPy) are used for signal processing and analysis.

  ✓ **Network Simulation and Protocol Development**: Languages like C, C++, and Python are used to develop network protocols and simulate telecommunication networks.

  ✓ **Automation and Control Systems**: Used in designing automated and control systems in telecommunications infrastructure.

  ✓ **IoT Development**: Programming languages such as C, Python, and JavaScript are used for developing IoT applications and devices.

✓ **DSP (Digital Signal Processing)**: Specialized languages and libraries are used to implement DSP algorithms for communication systems.

3. Types of C Programming IDEs

A computer programming Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE typically consists of a source code editor, build automation tools, and a debugger.

- **Cross-Platform IDEs**

✓ **Eclipse**: Widely used, supports multiple languages, extensible via plugins.

✓ **Code::Blocks**: Open-source, highly configurable, and supports multiple compilers.

✓ **NetBeans**: Modular IDE with support for C/C++ development.

- **Windows-Specific IDEs**

✓ **Microsoft Visual Studio**: Powerful IDE with extensive features for debugging and development.

✓ **Dev-C++**: Lightweight and simple, suitable for beginners.

- **Linux-Specific IDEs**

✓ **CodeLite**: Open-source IDE designed for C/C++ development.

✓ **KDevelop**: Integrated development environment for Linux users, supports various programming languages.

- **MacOS-Specific IDEs**

✓ **Xcode**: Apple's official IDE for macOS development, supports C/C++ among other languages.

- **Online IDEs**

✓ **Repl.it**: Web-based IDE supporting many languages, including C.

✓ **JDoodle**: Online compiler and IDE for C/C++ programming.

Practical Activity 1.1.2: Installation of C programming IDE

Task:

1: Referring to previous activity, you are requested to perform the given task.

John is an embedded computer programmer has a computer that lacks a C programming IDE, and he needs to write a firmware to be embedded in company hardware devices. To enable him to accomplish this, you are tasked with installing a C programming IDE on his computer and write a c program that display a message" hello architecture".

2: Using key readings **1.1.2,** List out procedures to be used to perform the given tasks

3: Install the c Programming IDE according the previous procedures in task 2.

4: Present the installed IDE to the whole class.

5: Ask clarification where necessary.

6: Perform the task provided in application of learning **1.1.**

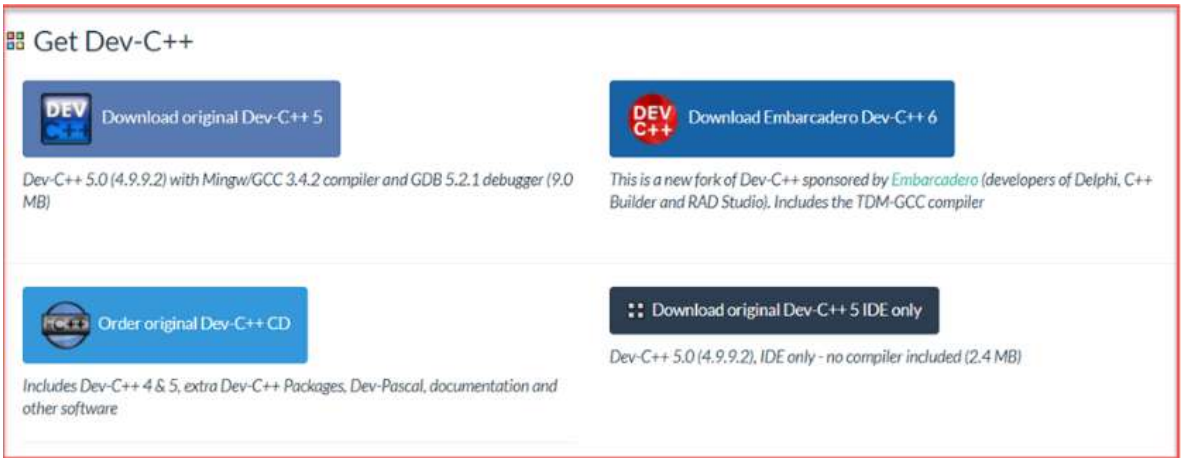
Key readings 1.1.2 Installation of C programming IDE

1. Installation of c programming IDE

- How to download setup .exe

There are many compilers available for C. You need to download any one. Here, we are going to use Dev-c++. It will work for both C and C++. To install the Turbo C software, you need to follow the following steps.

- ✓ Step 1) First you must download the Dev C++ on your Windows machine. Visit to Download Dev C++: http://www.bloodshed.net/
- ✓ Step 2) There are packages for different Operating Systems.



- ✓ Step 3) Under package Dev-C++ 5.0 (4.9.9.2) with Mingw/GCC 3.4.2 compiler and GDB 5.2.1 debugger (9.0 MB) Click on the link "Download from SourceForge".

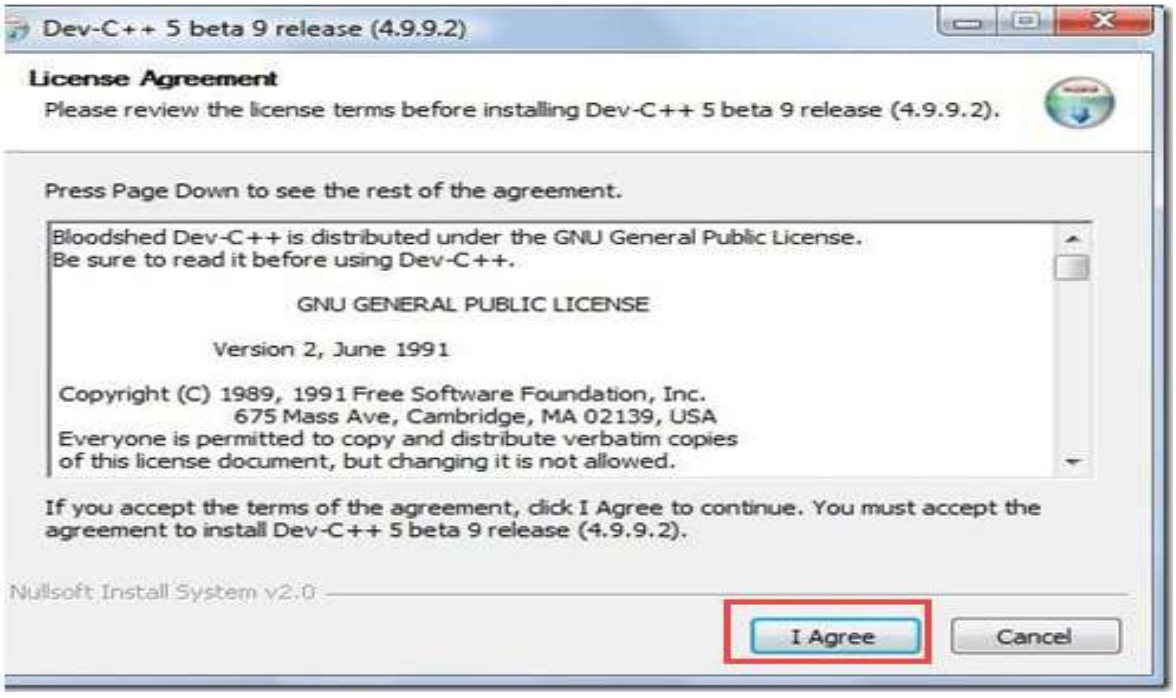
- ✓ Step 4) This package will download C++ .exe file for Windows that can be used to install on Windows 7/8/XP/Vista/10.

- ✓ Step 5) You will direct to SourceForge website, and your C++ download will start automatically.

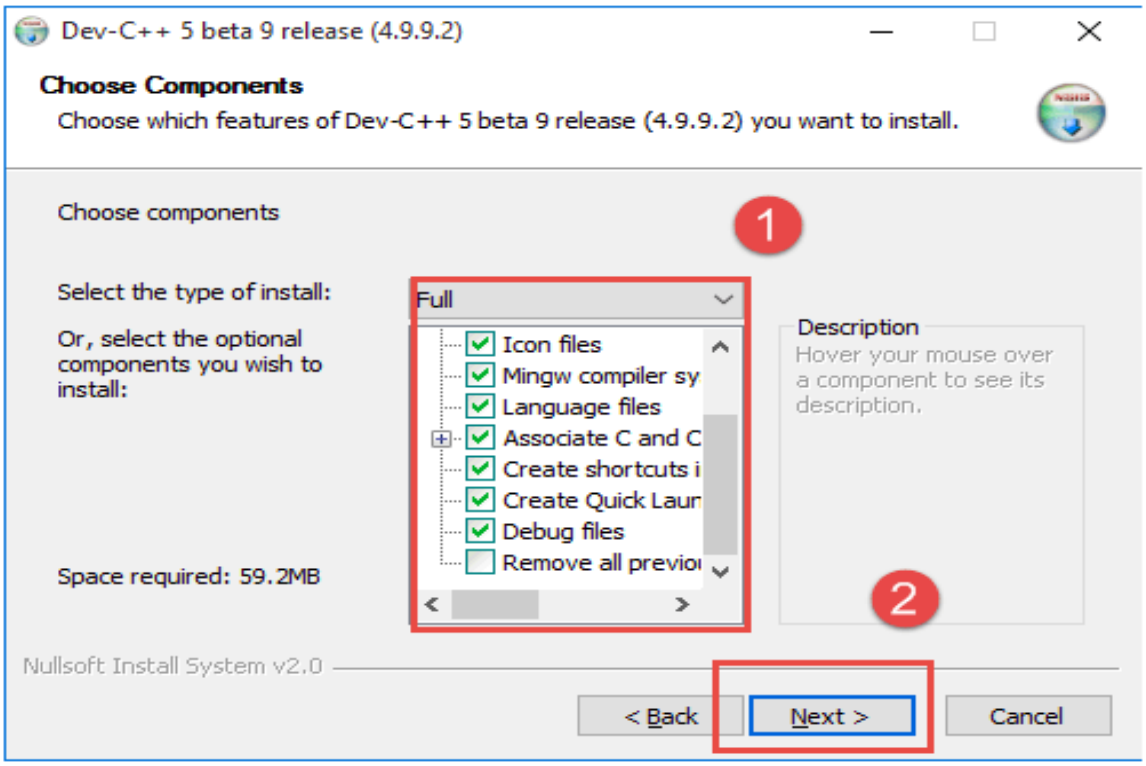
Click on save button to save. By default, it is saved in the “Downloads” folder.

- How to install setup.exe
- ✓ Step 1) Locate setup.exe file of c programming IDE
- ✓ Step 2) Double-click the .exe file. A dialog box will appear then click ok.

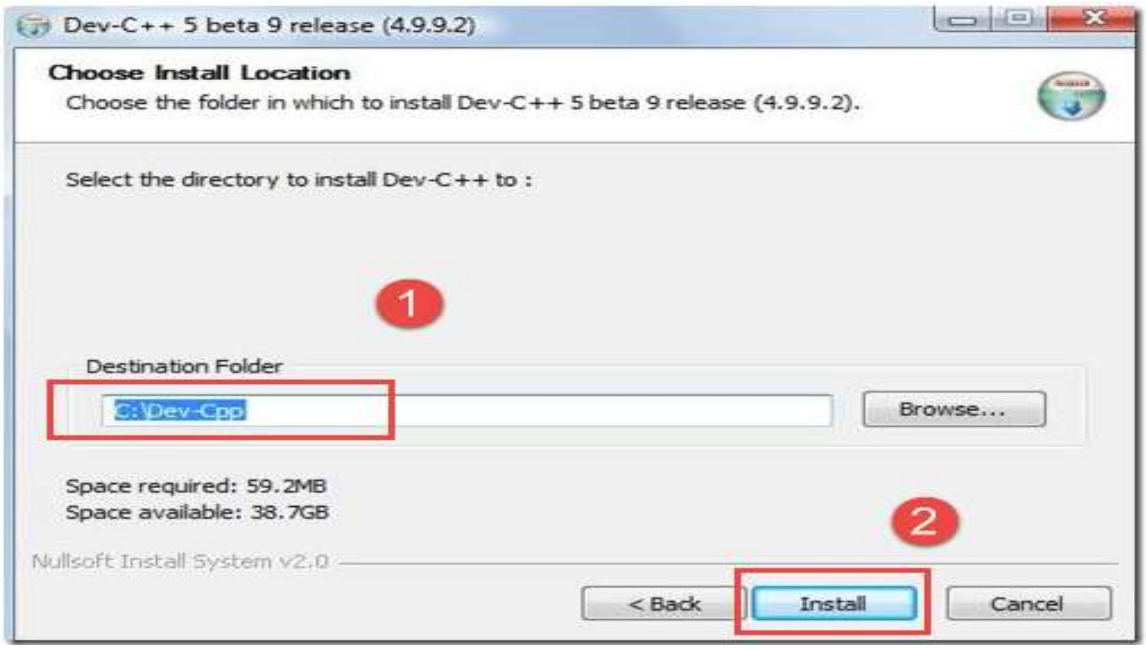
- ✓ Step 3) Then a screen for license agreement will appear. Click on “I agree” to proceed further.



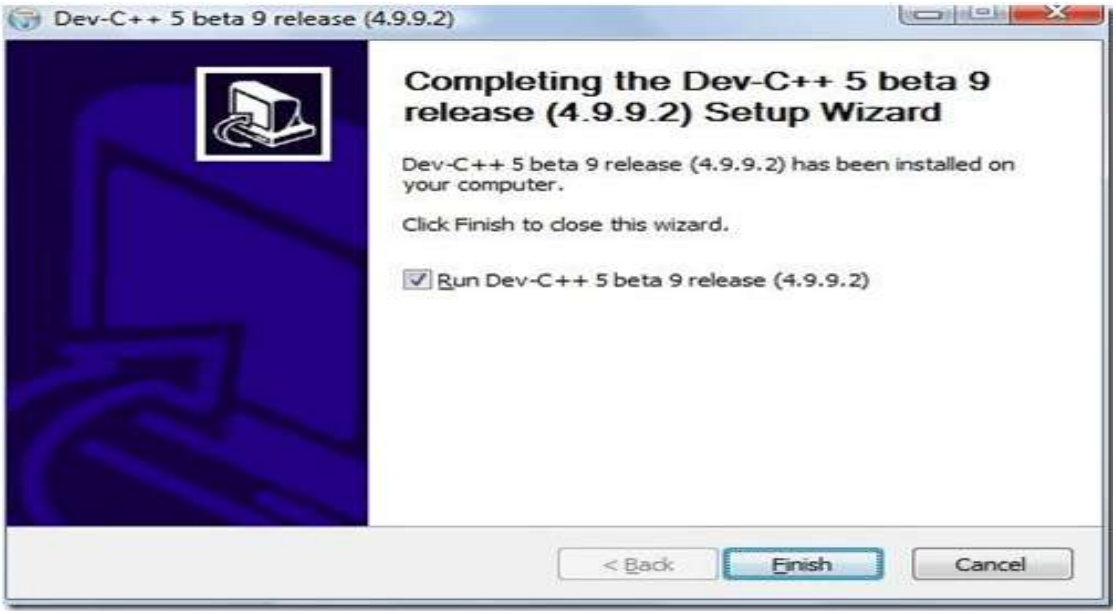
- ✓ Step 4) Just click on the “next” button.



✓ Step 5) Click on the "Install" button.



Now, Dev C++ is installed successfully on your Windows. Select " Run Dev C++" to run it and click on the " Finish" button.



2. Setting Up C Programming Environment

- **Installing a Compiler**
  - ✓ **GCC (GNU Compiler Collection)**: Available for Linux, Windows (via MinGW), and macOS.
  - ✓ **Clang**: A compiler for C language family, available for Linux, Windows, and macOS.
- **Choosing and Installing an IDE**
  - ✓ Select an IDE based on your operating system and preferences (e.g., Visual Studio for Windows, Eclipse for cross-platform use).
  - ✓ Download and install the chosen IDE from its official website.
- **Configuring the IDE**
  - ✓ **Setting Compiler Path**: Ensure the IDE knows the path to the installed compiler.
  - ✓ **Configuring Build and Debug Settings**: Set up the build configuration (debug/release), link libraries, and include directories.
  - ✓ **Writing and Running Code**: Create a new project, write C code, and use the IDE's build and run functionalities to compile and execute the code.
- **Using Command Line Tools**
  - ✓ **Compiling via Command Line**: Use commands like gcc filename.c -o outputfile to compile C programs.
  - ✓ **Running the Program**: Execute the compiled binary from the command line using ./outputfile.
- **Libraries and Dependencies**
  - ✓ **Standard Libraries**: Include standard libraries like stdio.h, stdlib.h, etc., in your programs.
  - ✓ **Third-Party Libraries**: Link any required third-party libraries during compilation.

Points to Remember

Types of programming Languages

- A computer programming language is a formal language used to write instructions that a computer can execute.
- There are two types: **low-level languages**, which are closely tied to hardware, and **high-level languages**, which offer more abstraction.
- Programming languages are essential in telecommunications for developing firmware and software for embedded systems, network monitoring, and automation.
- C Programming is a widely-used, easy-to-learn general-purpose language that helps understand core programming concepts, making it easier to learn other high-level languages.
- Integrated Development Environments (IDEs) are tools where programmers write code; C has various IDEs compatible with different operating systems like Windows, macOS, and Linux, with some being cross-platform.

Setup the environment if C programming IDE

- Visit the official website of the IDE (e.g., Visual Studio, Code::Blocks, Eclipse) and download the appropriate installer or Locate the setup of C programming IDE .
- Install a C compiler (e.g., GCC for Linux/Mac, MinGW for Windows) if the IDE doesn't include one.
- Follow the installation instructions provided by the installer, adjusting settings as needed.
- Set up the IDE to use the correct compiler and debugger.
- Configure paths for libraries and include files if necessary.
- Create a simple C program to ensure everything is working correctly.
- Compile and run the program to get the output.

Application of learning 1.1.

Madam Rose needs to write a C program to display her name, but her computer lacks a C programming IDE. you are requested install a C programming IDE on her computer.

Indicative content 1.2: Development of an Algorithm

Duration: 3 hrs

Theoretical Activity 1.2.1: Description of algorithms.

Tasks:

1: In small groups, you are requested to answer the following questions related to the description of algorithms.

i. What is an algorithm?
ii. What is pseudo code, and how is it used in the algorithm design process?
iii. Enumerates the different types of algorithms.

2: Provide the answer for the asked questions and write them on papers/flipchart or white board.

3: Present the findings/answers to the whole class

5: Ask for more clarification, where necessary.

4: For more clarification, read the key readings **1.2.1**

Key readings 1.2.1. Description of algorithms

1. Definition of an algorithm

Algorithm is the sequence of instructions which are involved in solving a given problem or accomplishment of a given task. It can be translated into a programming language in order to produce the result. A set of steps that generates a finite sequence of elementary computational operations leading to the solution of a given problem is called an algorithm.

- **Structure of an algorithm**

An algorithm is made mainly of the following parts:

✓ The variable declaration line
✓ The beginning of an algorithm
✓ The instruction's part
✓ The end

Example:

Var A as Integer
Start
A←5
End

Explanations:

Var A as Integer: is the variable declaration line

Start: marks the beginning of an algorithm

A←5: the instructions part

End: marks the end of an algorithm

2. Pseudo code

Artificial and informal language that helps programmers develop algorithms. Pseudo code is very similar to everyday English.

Pseudo code is a "text-based" detail (algorithmic) design tool. The rules of Pseudo code are reasonably straightforward. All statements showing "dependency" are to be indented. These include while, do, for, if, switch.

3. Types of algorithms

Algorithms are step-by-step procedures or formulas for solving problems, executing tasks, or achieving objectives. They can be classified into various types based on their design, structure, and purpose.

- **Sequential Algorithms**:

✓ Execute instructions in a linear sequence from start to finish.

✓ Example: Simple arithmetic operations.

- **Iterative Algorithms**:

✓ Repeat a sequence of instructions until a specific condition is met.

✓ Example: Searching or sorting algorithms like Bubble Sort or Binary Search.

- **Recursive Algorithms**:

✓ Call themselves with smaller instances of the same problem until reaching a base case.

✓ Example: Factorial calculation or Fibonacci sequence.

- **Divide and Conquer Algorithms**:

✓ Break down a problem into smaller, manageable sub-problems.

✓ Solve each sub-problem independently and combine the solutions.

✓ Example: Merge Sort or Quick Sort.

- **Greedy Algorithms**:

✓ Make locally optimal choices at each stage with the hope of finding a global optimum.

✓ Example: Minimum Spanning Tree algorithms like Prim's or Kruskal's.

- **Dynamic Programming Algorithms**:

✓ Solve complex problems by breaking them down into simpler sub-problems.

✓ Store solutions to sub-problems to avoid redundant calculations.

✓ Example: Fibonacci sequence using memorization.

- **Backtracking Algorithms**:

- ✓ Attempt to build a solution incrementally and backtrack when reaching a dead end.
- ✓ Example: N-Queens problem or Sudoku solving.
- **Randomized Algorithms**:
- ✓ Introduce randomness or probability in their execution.
- ✓ Example: Randomized Quick Sort or Monte Carlo algorithms.
- **Heuristic Algorithms**:
- ✓ Produce good solutions to problems that cannot be solved optimally.
- ✓ Often used in decision-making or optimization problems.
- ✓ Example: Genetic algorithms or Ant Colony Optimization.
- **Parallel Algorithms**:
- ✓ Execute multiple operations simultaneously to reduce execution time.
- ✓ Suitable for tasks that can be divided into independent sub-tasks.
- ✓ Example: Parallel Matrix Multiplication.
- **Importance of Algorithm Classification**
- ✓ **Problem-Specific Efficiency**: Different algorithms are suited to different types of problems based on their characteristics and constraints.
- ✓ **Optimization Goals**: Understanding algorithm types helps in choosing the most efficient solution approach.
- ✓ **Algorithm Design**: Helps in designing new algorithms or optimizing existing ones for specific tasks.

  Problem Solving: Provides a structured approach to solving computational problems across various domains.

Practical Activity 1.2.2: Developing an algorithm

Task:

1: Refer to previous activity 1.3.1, you are requested to perform the given task.

You are tasked with developing an algorithm and pseudocode to sort books on a library bookshelf by their titles alphabetically. The library currently has a disorganized shelf where books are placed randomly, and the librarian wants them arranged in alphabetical order for easier access and browsing by library patrons.

2: By using key readings **1.2.2**, list out procedures to be used to develop an algorithm.

3: write an algorithm using the procedures list above.

4: Present your work to whole class.

5: Ask clarification where necessary

Key readings 1.2.2: Developing an algorithm

1. Development of an algorithm

An **algorithm** is a step-by-step procedure or set of instructions to solve a specific problem or perform a specific task. When developing an algorithm, you need to break down the problem into smaller, manageable steps that lead to the desired solution.

- **Understanding the Problem:**

✓ **Define the Problem:** Clearly understand and define the problem you are trying to solve. Gather all necessary information and requirements.

✓ **Identify Inputs and Outputs:** Determine what inputs are needed and what outputs are expected. This helps in designing the steps required to transform inputs into outputs.

- **Breaking Down the Problem:**

✓ **Decomposition:** Break down the problem into smaller, manageable parts. Each part should be easier to handle and solve.

✓ **Modularity:** Design the algorithm in a modular way, where each module or function handles a specific part of the problem.

- **Designing the Algorithm:**

✓ **Step-by-Step Instructions:** Outline the steps needed to solve the problem in a logical sequence. Each step should be clear and unambiguous.

✓ **Flow Control:** Determine the control structures needed, such as loops, conditionals, and function calls, to manage the flow of the algorithm.

- **Writing Pseudocode:**

✓**Informal Language:** Use pseudocode, which is a mixture of natural language and programming concepts, to outline the algorithm. Pseudocode helps in planning the algorithm without worrying about syntax.

✓**Clarity:** Ensure the pseudocode is easy to understand. It should be simple enough for someone without programming knowledge to follow the logic.

- **Analysing the Algorithm:**

✓ **Efficiency:** Analyze the algorithm for time and space complexity. Consider how the algorithm scales with increasing input size.

✓ **Correctness:** Verify that the algorithm correctly solves the problem for all possible inputs. Consider edge cases and test the algorithm with various scenarios.

2. Examples

1. **Formulate an Algorithm to Check if a Number is Prime.**

- **Problem Understanding:**

✓ Input: A number n.

✓ Output: True if n is prime, False otherwise.

- **Steps:**

✓ If n <= 1, return False.

✓ Loop from 2 to the square root of n.

✓ If n is divisible by any number in this range, return False.

✓ If no divisor is found, return True.

- **Pseudocode:**

Algorithm IsPrime(n)

1. If n <= 1 then

a. Return False

2. For i from 2 to sqrt(n):

a. If n % i == 0 then

i. Return False

3. End loop

4. Return True

End Algorithm

2. You are given a list of numbers. Develop an algorithm to find the largest number in the list.

3. Step-by-Step Development of the Algorithm:

1. **Understand the Problem:**

✓ We need to find the largest number from a list of numbers.

✓ Input: A list of n numbers (e.g., [3, 5, 7, 2, 8]).

✓ Output: The largest number in the list (e.g., 8).

2. **Identify Key Operations:**

✓ Iterate through the list of numbers.

✓ Compare each number to find the largest one.

3. **Outline the Steps:**

Step 1: Initialize a variable max to store the largest number. Start by setting it to the first number in the list.

Step 2: Loop through the list starting from the second number.

Step 3: For each number, compare it with max. If the number is greater than max, update max with this number.

Step 4: Continue the loop until all numbers have been checked.

Step 5: The value in max at the end of the loop is the largest number.

Step 6: Output the value of max.

4. **Translate into Pseudocode:**

Algorithm FindLargestNumber(List)

1. Set max = List[0]

2. For each number in List starting from the second number:

a. If number > max then

i. Set max = number

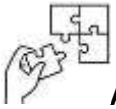
3. End loop
4. Output max
End Algorithm

Points to Remember

An algorithm is a sequence of instructions designed to solve a problem or achieve a specific task. It can be translated into a programming language to produce a desired result.
Structure of an Algorithm:

- Variable Declaration: Defines variables used in the algorithm.
- Beginning: Marks the start of the algorithm.
- Instructions: Contains the sequence of steps to be executed.
- End: Marks the completion of the algorithm.
- Types of Algorithms: Sequence, Branching (Selection), Loop (Repetition):

- **Pseudo Code:**
  - ✓ An informal, text-based language for designing algorithms.
  - ✓ Helps programmers plan and outline solutions before coding.
  - ✓ Rules are straightforward and aid in algorithmic design.
- **Develop an algorithm**
  - ✓ Obtain detailed information on the problem.
  - ✓ Analyse the problem.
  - ✓ Think of a problem-solving approach.
  - ✓ Review the problem-solving approach and try to think of a better Alternative.
  - ✓ Develop a basic structure of the Algorithm.
  - ✓ Optimize, Improve and refine.
- **Write the pseudo code**
  - ✓ Always capitalise the initial word (often one of the main six constructs).
  - ✓ Make only one statement per line.
  - ✓ Indent to show hierarchy, improve readability, and show nested constructs.
  - ✓ Always end multi-line sections using any of the END keywords (ENDIF, ENDWHILE, etc.).
  - ✓ Keep your statements programming language independent?
  - ✓ Use the naming domain of the problem, not that of the implementation. For instance: “Append the last name to the first name” instead of “name = first+ last.”
  - ✓ Keep it simple, concise and readable.



Application of learning 1.2.

The GB company is located in Kigali, they pay their employees using paper. Due to this issue the employees did not get easily their payroll statement. As a programmer Write an algorithm and pseudocode to calculate and display the total monthly expenses based on various categories like rent, groceries, utilities, and entertainment. The user should input expenses for each category.

Indicative content 1.3: Development of a flowchart

Duration: 2 hrs

Theoretical Activity 1.3.1: Description of the flowchart

Tasks:

1: In small groups, you are requested to answer the following question related to the description of the flowchart symbols.

i. What is a flowchart?
ii. What are the importances of a flowchart in visualizing processes.
iii. What are symbols used to design a flowchart

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: In addition, ask questions where necessary.

5: For more clarification, read the key **readings 1.3.1**

Key readings 1.3.1. Description of the flowchart

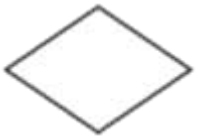
Description of Flowchart

- **Definition of Flowchart**

  Flowchart is a graphical representation of the sequence of operations in an information system or program. Information system flowcharts show how data flows from source documents through the computer to final distribution to users. Program flowcharts show the sequence of instructions in a single program or subroutine. Different symbols are used to draw each type of flowchart.

  Flowchart symbols

Name of the symbol	Diagram	Description
Oval		Denotes the start and end of the flowchart
Rectangle		A calculation or assigning of value to a

		variable.eg.Addition,subtraction,division, etc
Rhombus or Diamond		Denotes a decision (branch)to be made.The program should continue along one of two routines.True of False
Circle		Can be used to eliminate lengthy of flowrines. It's used to indicate that one symbol is connected to another.
Parallelogram		Denotes an Input operations
Flow Line		Lines indicate the sequence of steps and the direction of flow. The flow is assumed to go from top to bottom and from left to right.

Practical Activity 1.3.2: Designing a flowchart

Task:

1: Referring to previous activities (1.2.1) you are requested to perform the given task. Using a flowchart Describe the steps involved in going to the cinema to watch a movie, including actions like choosing a film, buying tickets, and making decisions, such as verifying showtime availability.

2: From key readings **1.3.2**, list out procedures to be used to perform the given task f

3: Perform the given tasks according to the procedures listed in task 2.

4: Present your work to the whole class.

5: Ask clarification where necessary

Key readings 1.3.2: Designing a flowchart

1. Design a flowchart

A flowchart is a visual representation of a process, algorithm, or system that outlines the steps or actions involved in a sequence.

- **Develop a flowchart.**

steps to develop a flowchart according to the given problem

Step 1: Know the purpose of your flowchart.

Before knowing how to draw a flowchart, it is mandatory to find out why you're creating one in the first place. Your flowchart can have various goals, such as:

- ✓ Making a complex process easier to understand
- ✓ Improving an existing business process by identifying bottlenecks
- ✓ Explaining or communicating a process to someone else
- ✓ Standardizing a process for consistency and efficiency

When you know the purpose of your flowchart, it's easier to choose the type of flowchart to create, pick the most relevant template, and know whether to focus more on design or function when making one.

Step 2: Start with a template.

Creating a flowchart from scratch can be intimidating. Luckily, you don't have to. There are plenty of flowchart templates available online that you can customize and adapt for your own use. Try to select a template that's created specifically for your field or industry so it takes you less time and effort to finalize.

Step 3: Add shapes and symbols.

Shapes and symbols play a vital role in any flowchart. They define the components of your flowchart, and each symbol represents something unique. After you've identified the purpose of your flowchart, it's time to draw your diagram. The first thing you need to do here is add the symbols that represent all the components in your flowchart.

Step 4: Connect your shapes with lines and arrows.

Once you've added all the different shapes you need, it's time to connect each one with lines and arrows. We recommend using arrows if you want to show a specific direction of your flow of information. For example, the flowchart example below

visualizes how information is flowing in different directions with the help of arrows.

Step 5: Split paths or add decisions.

In some cases, you may want to split paths and add decisions to your flowchart, such as adding two different paths based on Yes/No decisions. This is useful for visualizing complex processes that require people to take different actions based on varying inputs and outputs.

Step 6: Customize your flowchart's appearance.

The next step is to customize your flowchart's appearance. Your flowchart's design will depend on how you plan to use your diagram, where you'll use it and who the audience is. First, you may want to change the size, color, border, roundness and other characteristics of your shapes, symbols and lines.

Step 7: Download or share your flowchart.

Once your flowchart is ready, it's time to download it or share it with your target audience.

Points to Remember

- Flowcharts are diagrammatic representations of processes.
- Flowchart Symbols ovals, rectangles, diamonds, and arrows to show the sequence of steps, decision points, and outcomes. They simplify complex processes, enhance communication, and aid in problem-solving, documentation, and training.

Design a flowchart

- Define the Objective
- Use Standard Symbols
- Maintain Logical Flow

Application of learning 1.3.

In southern province different sectors want to construct the playground for each school. They need to calculate the area of each ground using its length and width. Draw a flowchart that will accept the length, width to calculate and print the area of the playground. (area = length * width).

Learning outcome 1 end assessment

Theoretical assessment

1. What is a symbol in a flowchart represents the start and end of the process?
2. What is a programming language?
3. Explain the purpose of a flowchart in software development.
4. Describe the steps you would follow to install a C programming IDE like Code::Blocks on your computer.
5. Compare and contrast the use of procedural programming and object-oriented programming paradigms in software development.
6. Evaluate the pros and cons of using an interpreted language like Python for developing a web application.
7. Name two programming languages commonly used for embedded systems development.
8. What is the primary function of pseudo code in the development process?
9. Given a scenario where you need to automate a repetitive task in a telecommunications system, which programming language would be most appropriate and why?
10. Analyze the use of flowcharts versus pseudo code in a team setting for a large-scale project. Which would you prefer and why?
11. Copy and complete the following table

Name of the symbol	**Diagram**	**Description**
Oval	……	……
……	……	It is used for data processing
……		……
Circle	……	……
……		……

12. Compare and contrast different programming paradigms (procedural, object-oriented, functional, and logic programming). Provide examples of languages that exemplify each paradigm and discuss their primary use cases.
13. Describe the differences between compiled and interpreted programming languages. Discuss the advantages and disadvantages of each approach with examples of languages that use these methods.

Choose the letter corresponding to the correct answer:

14. Which of the following is an advantage of compiled languages over interpreted languages?

a) Easier to debug
b) Faster execution
c) Platform independence
d) Immediate feedback

15. Which of the following steps would be the best starting point for creating a new software application?

a) Writing the final code
b) Designing a flowchart or pseudo code
c) Debugging the code
d) Compiling the program

16. In what scenario would functional programming be preferred over logic programming?

a) Concurrent processing tasks
b) Artificial intelligence applications
c) Designing embedded systems
d) Web development

17. Which IDE would you choose for C/C++ development if you needed a lightweight, customizable environment?

a) Eclipse
b) Visual Studio
c) Code::Blocks
d) CLion

18. Pseudo code is primarily used for:

a) Debugging code
b) Writing final programs
c) Describing the steps in an algorithm
d) Designing hardware

Practical assessment

1. You need to write an algorithm and pseudocode for creating a grocery shopping list. The program should allow the user to input items they need to buy, then display the complete list of items.
2. Develop an algorithm and pseudocode for a library management system to add and display books. The user can enter the book's title and author, and the system should store and display the list of books.
3. Create an algorithm and pseudocode for a simple calculator that performs addition, subtraction, multiplication, and division based on user inputs.

References

1997 ISBN : Table of Contents Getting Help in an Integrated Development Environment," no. August, pp. 1–70, 1997.

S. G. Kochan, *Programming in C Warning and Disclaimer Bulk Sales*. 2004.

S. C. Dewhurst and K. Stark, *Programming in C++*, vol. 2, no. 4. 1991. doi: 10.1145/126983.126989.

M. Vine, *C Programming for the Absolute Beginner*. 2008.

https://www.scribd.com/presentation/413537916/Lect1-Algorithms-and-Flowchart-ppt?utm_medium=cpc&utm_source=google_pmax&utm_campaign=3Q_Google_Performance-Max_RoW&utm_term=&utm_device=c&gclid=CjwKCAjw36GjBhAkEiwAKwIWyYBOAwSwzUDmki-xCos35zYEx-s7uQqw4Q2rHSpf2FG2VySd7fYxMRoCmJAQAvD_BwE#

https://terrorgum.com/tfox/books/flowchartandalgorithmbasics.pdf

https://www.w3resource.com/c-programming-exercises/conditional-statement/index.php

https://faradars.org/wp-content/uploads/2015/07/algorithm-and-flow-chart.pdf


evaluated
statement
standard
case
void
contains
array
constant
variables
function
POINTER
size
zero
define
int
read
char
USED
CHARACTER
INITIALIZED
PROGRAM
include
TYPE
return
CALLED
operator
double
structure
time
write
String
specifier
SPACE
STRUCT
input
error
library
RESULT
object
expression
declaration
external
NULL

Indicative contents
2.1. Description of C programming concepts **2.2. Description of C program structure** **2.3. Application of condition statements** **2.4. Application of loops** **2.5. Application of functions** **2.6. Application of pointers** **2.7. Application of Arrays**

Key Competencies for Learning Outcome 2: Write C Programming Codes

Knowledge	Skills	Attitudes
• Description of C structure • Description of control structure • Differentiating interpreter from compiler • Differentiate statement from expression • Differentiation variable from constants • Differentiation different data types • Description of functions • Identification of the scope of a variable	• Installing C based IDE • Applying variables • Using Operators • Using different data types. • Applying conditional statements. • Applying functions • Applying control structure • Applying Loops • Applying Arrays • Document program • Using comments	• Having Critical thinking • Being Passionate • Having Team work spirit • Have Persistence • Being Hard working • Being Self-motivated • Having Self confidence

Duration: 60 hrs

Learning outcome 2 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly the building blocks as per C programming language

2. Describe correctly the C program structure as per functional program

3. Apply properly a condition statements based on a syntax and certain conditions.

4. Apply properly C loops based on syntax and certain instructions.

5. Apply properly C functions based on syntax and specified task.

6. Apply properly C programming pointer based on a given problem

7. Apply properly C Arrays based on syntax, data store and their use.

Resources

Equipment	Tools	Materials
• Computer	• Dev-C++	• Electricity • Internet

Indicative content 2.1: Description of C programming concepts

Duration: 10 hrs

Theoretical Activity 2.1.1: Description of C Tokens

Tasks:

1: In small groups, you are requested to answer the following questions related to the description of resonant circuits.

i. What is a token in the context of the C programming language?
ii. How are tokens used to represent fundamental building blocks in C code?
iii. Name and describe the five main categories of C tokens.
iv. What are identifiers in C, and how do they differ from keywords?
v. Give examples of valid and invalid identifiers in C.
vi. Explain the role of keywords in C and provide some examples.
vii. Name the three primary types of constant tokens in C.

2: Provide the answer for the asked questions and write them on papers.
3: Present the findings/answers to the whole class
4: For more clarification, read the key **readings 2.1.1.**
5: In addition, ask questions where necessary.

Key readings 2.1.1. Description of C Tokens

1. C Tokens

In C programming, tokens are the smallest units of a program that have meaningful interpretation by the compiler. These tokens form the basic building blocks of C syntax. Here are the main types of tokens in C:

- **Keywords:**

✓ Keywords are reserved words that have predefined meanings in C. They cannot be used as identifiers (variable names, function names, etc.).
✓ Examples: int, float, if, else, for, while, return, void, switch, case, break, continue, sizeof, typedef, struct, union, enum, const, static, extern, volatile, auto, register, default, do, double, short, long, signed, unsigned, char, goto.

2. Identifiers:

✓ Identifiers are names given to variables, functions, arrays, structures, etc., by the programmer.

- ✓ Rules for identifiers:
- Must start with a letter (A-Z or a-z) or an underscore (_).
- Can be followed by letters, digits (0-9), or underscores.
- Case-sensitive.
- Cannot be a keyword.
- ✓ Example: main, sum, counter, MAX_SIZE, is_valid.

3. Constants:

- ✓ Constants are fixed values that do not change during program execution.
- ✓ **Types of constants:**
- ✓ **Integer Constants:** Whole numbers without decimal points.
- Example: 123, -456, 0, 0xFF (hexadecimal).
- ✓ **Floating-point Constants:** Numbers with decimal points.
- Example: 3.14, -0.5, 2.0e3 (exponential).
- ✓ **Character Constants:** Single characters enclosed in single quotes.
- Example: 'A', '7', '\n' (newline character).
- ✓ **String Literals:** Sequences of characters enclosed in double quotes.
- Example: "Hello, World!", "C programming".

- **String and Character Literals:**

- ✓ String literals are sequences of characters enclosed in double quotes (").
- ✓ Character literals are single characters enclosed in single quotes (').
- ✓ Example: "Hello", 'A', "123", '\n'.

- **Operators:**

- ✓ Operators perform operations on variables and values.
- ✓ Types of operators:
- Arithmetic Operators: +, -, *, /, %.
- Relational Operators: ==, !=, <, >, <=, >=.
- Logical Operators: &&, ||, !.
- Bitwise Operators: &, |, ^, ~, <<, >>.
- Assignment Operators: =, +=, -=, *=, /=, %=, <<=, >>=, &=, |=, ^=.
- Increment and Decrement Operators: ++, --.
- Conditional (Ternary) Operator: ? :.
- Address and Indirection Operators: &, *.
- ✓ Example: a + b, x < y, !flag, ~mask, a = b, x && y.

- **Punctuation Symbols:**

- ✓ Punctuation symbols are special characters that serve as separators or delimiters in C.
- ✓ Examples: , (comma), ; (semicolon), : (colon), . (dot), () (parentheses), {} (braces), [] (brackets).

- **Whitespace:**

- ✓ Whitespace includes spaces, tabs, and newline characters.

- ✓ Whitespace separates tokens but is otherwise ignored by the compiler.

5. Comments:

- ✓ Comments provide information to the programmer and are ignored by the compiler.
- ✓ Types of comments:
- **Single-line comments:** // This is a comment.
- **Multi-line comments:** /* This is a multi-line comment */.

Practical Activity 2.1.2: Writing the Data types in a c program

Task:

1: Read the following task. you are requested to perform the given task.

John, a computer science student, is tasked with developing a program that manages inventory for a small retail store. His program needs to handle different types of data, from item quantities to prices, and ensure accurate calculations and storage of information. Write all data types John need to complete his task.

2: Follow the instructions in key readings **2.1.2** to perform the given task

3: write down the findings from the discussions.

4: Present the finding to whole class.

5: Ask clarification where necessary

6: Read key reading **2.1.2**

Key readings 2.1.2: Writing the Data types in a c program

1. Data type in c programming

In C programming, data types are fundamental for defining the type of data that variables can hold and manipulate. Implementing data types effectively ensures proper storage, manipulation, and representation of data within a program. Here's an overview of how data types are implemented in C programs:

- **Choosing Appropriate Data Types:**
- ✓ **Integer Types (int, short, long):** Used for storing whole numbers. The choice depends on the range of values needed (e.g., int for general purposes, short for small integers, long for larger integers).
- ✓ **Floating-Point Types (float, double, long double):** Used for handling numbers with decimal points. float provides single-precision, double offers double-precision, and long double provides extended precision.

- ✓ **Character Type (char):** Used for storing single characters. It can also be used to represent small integers within a limited range.
- ✓ **Derived and User-Defined Types:** Structs, unions, enums, and typedefs allow programmers to define custom data types that encapsulate multiple values or create symbolic names for existing types.

- **Variable Declaration and Initialization:**

- ✓ Variables must be declared with a specific data type before they can be used. For example:

```
int age;
float price;
char initial;
```

- ✓ Initialization assigns an initial value to a variable at the time of declaration:

```
int quantity = 10;
float pi = 3.14;
char grade = 'A';
```

- **Operations and Expressions:**

- ✓ Data types determine how operations are performed and how values are interpreted:
- Arithmetic operations handle numerical data types (int, float, double).
- Relational and logical operations compare values based on their data types (int, float, char).
- Bitwise operations manipulate binary representations of integers (int, char).

- **Type Casting:**

- ✓ Type casting allows conversion of one data type to another:
- Implicit casting occurs automatically by the compiler when types are compatible.
- Explicit casting involves the programmer specifying the conversion, which can help avoid loss of data or precision.

- **Memory Allocation:**

- ✓ Data types influence the amount of memory allocated to variables:
- Sizes of data types vary based on the compiler and system architecture (int typically 4 bytes, float 4 bytes, double 8 bytes).
- Arrays and structs allocate contiguous memory based on the size and alignment requirements of their elements.

- **Input and Output Handling:**

- ✓ Functions like scanf and printf are used for input and output operations, respecting the data type of variables:

```
int age;
```

```
printf("Enter your age: ");
scanf("%d", &age);
```

- **Error Handling and Validation:**

✓ Proper data type usage helps in error prevention and ensures that operations are performed correctly:

- Validate user inputs to ensure they match expected data types and constraints.
- Handle exceptions and errors that arise from incompatible data types or improper conversions.

Theoretical Activity 2.1.3: Description Variables in C program

Tasks:

1: Read carefully and answer the following questions:

i. What is a variable in c programming?
ii. Name the three fundamental variables in C.
iii. what is the Rules to write variable names
iv. Explain the differences between local, global and static variables in C.
v. What is the syntax for declaring and initialising variables in C ?
vi. Explain the importance of initialising variables when declaring them.

2: Write down the founded answer from your group
3: Present the findings to class
4: Follow the expert view from the trainer
5: Ask for more clarification where necessary
6: Read the key reading **2.1.3**

Key readings 2.1.3.: Description Variables in C program

1. Variables in c programming

A variable is a name of memory location. It is used to store data. Variables are changeable, we can change the value of a variable during execution of a program. It can be reused many times.

Note: Variable are nothing but identifiers

- **Rules to write variable names:**

✓ A variable name contains a maximum of 30 characters/ Variable name must be up to 8 characters.

✓ A variable name includes alphabets and numbers, but it must start with an alphabet.

✓ It cannot accept any special characters, blank spaces except underscore(_).

✓ It should not be a reserved word.

✓ **Declaration of Variables:**

A variable can be used to store a value of any data type. The declaration of variables must be done before they are used in the program. The general format for declaring a variable.

- **Syntax :**

data_type variable-1,variable-2,----- , variable-n;

Variables are separated by commas and the declaration statement ends with a semicolon.

Example: int x,y,z;

float a,b;

char m,n;

Assigning values to variables : values can be assigned to variables using the assignment operator (=). The general format statement is :

- **Syntax**

variable=constant;

Ex : x=100;

a= 12.25;

m="f";

we can also assign a value to a variable at the time the variable is declared. The general format of declaring and assigning value to a variable is :

- **Syntax**

data_type variable =constant;

Example: int x=100;

float a=12.25;

char m="f";

Types of Variables in C

2. There are many types of variables in c:

- **Local variable**
  Scope: Local variables are declared within a specific block of code, typically within a function or a compound statement. They are only accessible within the block where they are defined.
  Lifetime: Local variables have a limited lifetime. They are created when the program enters the block in which they are defined and destroyed when the block exits.

```
void myFunction() {
  int localVar = 42; // localVar is a local variable
  // Code here }
```

- **Global variable**
  Scope: Global variables are declared outside of any function, typically at the top of a C source file. They are accessible from any function within the entire program.
  Lifetime: Global variables have a program-wide lifetime. They are created when the program starts and exist until the program terminates.

```
int globalVar = 100; // globalVar is a global variable
void myFunction() {
  // Access globalVar here }
```

- **static variable**
  Scope: Static variables can be either local or global, depending on where they are declared. Local static variables are declared within a function, but they retain their value between function calls. Global static variables are declared at the global scope and are accessible within the entire program.
  Lifetime: Static variables have an extended lifetime. Local static variables retain their value between function calls, while global static variables exist for the duration of the program.

```
void myFunction() {
  static int staticVar = 0; // staticVar is a local static variable
  // staticVar retains its value between function calls
  staticVar++; }
```

Theoretical Activity 2.1.4: Description of constants

Tasks:

1: Read carefully and answer the following questions

i. What is a constant in C programming?
ii. Why are constants used in programming, and what benefits do they offer?
iii. Name and describe the three main types of constants in C.
iv. Explain the format and syntax of integer constants in C.
v. Describe how floating-point constants are represented in C.

2: Discuss on the given topic and write down their finding.

3: Present their finding to the whole class

4: Ask clarification if necessary

5: Read key reading **2.1.4**

Key readings 2.1.4. Description of constants

1. Constants in C Programming

In C programming, **constants** are fixed values that the program cannot alter during its execution. Constants are used to define values that remain the same throughout the program, making the code more readable and easier to maintain. Different types of constants in C:

- **Integer Constants**
  - ✓ **Decimal Constants:** Base 10 numbers (e.g., 100, -42).
  - ✓ **Octal Constants:** Base 8 numbers, prefixed with 0 (e.g., 071, -052).
  - ✓ **Hexadecimal Constants:** Base 16 numbers, prefixed with 0x or 0X (e.g., 0x1A, 0X7F).
- **Floating-point Constants**
  - ✓ Represent real numbers (numbers with a fractional part).
  - ✓ Can be written in decimal form (e.g., 3.14, -0.001) or in exponential form (e.g., 2.5e3 for 2500).
- **Character Constants**
  - ✓ A single character enclosed within single quotes (e.g., 'A', '9', '#').
  - ✓ Represented by their ASCII values in memory (e.g., 'A' is stored as 65).
- **String Constants**
  - ✓ A sequence of characters enclosed within double quotes (e.g., "Hello, World!").
  - ✓ The compiler automatically adds a null character ('\0') at the end of the string to indicate its termination.
- **Enumeration Constants**

- ✓ Constants defined using the enum keyword, representing integral constants (e.g., enum days {SUNDAY, MONDAY, TUESDAY}; where SUNDAY is 0, MONDAY is 1, and so on).
- **Symbolic Constants**
- ✓ Defined using the #define preprocessor directive (e.g., #define PI 3.14159).
- ✓ These are replaced by their corresponding values during the preprocessing stage of compilation.

Theoretical Activity 2.1.5: Description of operators

Tasks:

1: Read carefully and answer the following questions

i. What are operators in programming?

ii. Describe the types of operators in programming

2: Discuss on the given topic and write down their finding.

3: Present their finding to the whole class

4: Ask clarification if necessary

5: Read key reading **2.1.5**

Key readings 2.1.5. Description of operators

1. Operators in c programming

In C programming, operators are symbols that perform operations on variables and values. Here is a list of the most common types of operators in C:

- **Arithmetic Operators**

These operators are used to perform basic arithmetic operations.

- ✓ + : Addition
- ✓ - : Subtraction
- ✓ * : Multiplication
- ✓ / : Division
- ✓ % : Modulus (remainder after division)

- **Relational Operators**

These operators are used to compare two values or expressions.

- ✓ == : Equal to
- ✓ != : Not equal to
- ✓ > : Greater than
- ✓ < : Less than

- ✓ >= : Greater than or equal to
- ✓ <= : Less than or equal to

- **Logical Operators**

  These operators are used to combine multiple conditions or expressions.

- ✓ && : Logical AND (true if both operands are true)
- ✓ || : Logical OR (true if at least one operand is true)
- ✓ ! : Logical NOT (inverts the truth value)

- **Bitwise Operators**

  These operators are used to perform operations at the bit level.

- ✓ & : Bitwise AND
- ✓ | : Bitwise OR
- ✓ ^ : Bitwise XOR (exclusive OR)
- ✓ ~ : Bitwise NOT
- ✓ << : Left shift
- ✓ >> : Right shift

- **Assignment Operators**

  These operators are used to assign values to variables.

- ✓ = : Simple assignment
- ✓ += : Add and assign
- ✓ -= : Subtract and assign
- ✓ *= : Multiply and assign
- ✓ /= : Divide and assign
- ✓ %= : Modulus and assign
- ✓ &= : Bitwise AND and assign
- ✓ |= : Bitwise OR and assign
- ✓ **^= :** Bitwise XOR and assign
- ✓ **<<=:** Left shift and assign
- ✓ **>>=:** Right shift and assign

6. Unary Operators

These operators operate on a single operand.

- ✓ **+ :** Unary plus (indicates positive value, usually optional)
- ✓ **- :** Unary minus (negates an expression)
- ✓ **++ :** Increment (increases the value by 1)
- ✓ **-- :** Decrement (decreases the value by 1)
- ✓ ! : Logical NOT
- ✓ ~ : Bitwise NOT

- **Ternary Operator**

  This operator takes three operands and is used for making decisions.

- ✓ **? : :** Ternary conditional (e.g., condition ? value_if_true : value_if_false)

- **Special Operators**

- ✓ **sizeof :** Returns the size of a variable or data type (e.g., sizeof(int))
- ✓ **& :** Address-of operator (returns the memory address of a variable)
- ✓ *** :** Dereference operator (used with pointers to access the value at a particular address)
- ✓ **-> :** Structure pointer operator (used to access members of a structure through a pointer)
- ✓ **. :** Structure member operator (used to access members of a structure)

Points to Remember

C Tokens

In C programming, tokens are fundamental units that the compiler interprets. They encompass keywords, identifiers, constants, operators, punctuation symbols, whitespace, and comments, serving as the basic building blocks of C syntax:

- **Keywords:** Reserved words with predefined meanings, like int, float, if, else, etc.
- **Identifiers:** User-defined names for variables, functions, etc., following specific rules and case-sensitive.
- **Constants:** Fixed values like integer (123), floating-point (3.14), character ('A'), and string literals ("Hello").
- **Operators:** Symbols performing operations (arithmetic, relational, logical, bitwise, assignment, etc.) on variables.
- **Punctuation Symbols:** Special characters (, ,, ;, : :, (), {} {}, []) that structure C code.
- **Whitespace:** Spaces, tabs, newlines used for token separation, ignored by the compiler.
- **Comments:** Provide non-executable information to programmers (// Single-line, /* Multi-line */).

C variables

- **Choosing Appropriate Data Types**: Selecting the right data type (int, float, char, etc.) based on the nature and range of the data ensures proper storage and manipulation.
- **Variable Declaration and Initialization**: Variables must be declared with a specific data type, and initializing them at the time of declaration ensures they start with a defined value.
- **Operations and Expressions**: Data types dictate how operations (arithmetic, relational, logical, bitwise) are performed and how values are interpreted.
- **Type Casting**: Type casting allows conversion between data types, with implicit casting done automatically and explicit casting requiring programmer intervention.
- **Memory Allocation**: Data types determine the amount of memory allocated to variables, affecting how arrays and structs manage contiguous memory.

- **Variable Naming Rules**: Variable names must start with an alphabet, can include numbers, but cannot contain special characters or spaces (except the underscore) and should not be a reserved word.
- **Declaration of Variables**: Variables must be declared before use, specifying the data type (e.g., int x;).
- **Assigning Values**: Values can be assigned to variables either during or after declaration using the assignment operator (=).
- **Types of Variables**: Variables in C can be local, global, or static, each with different scopes and lifetimes.

Local vs. Global Variables: Local variables are accessible only within the block they're declared in, while global variables are accessible throughout the entire program

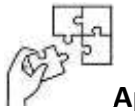
Constants in C Programming

Constants are fixed values that do not change during the execution of a program, and they are also known as literals. C programming supports several types of constants:

- Types of C Constants
  - ✓ Integer Constants
  - ✓ Character Constants
  - ✓ String Constants

C programming operators

- ✓ **Arithmetic Operators** (+, -, *, /, %) for basic math operations.
- ✓ **Relational Operators** (==, !=, >, <, >=, <=) for comparing values.
- ✓ **Logical Operators** (&&, ||, !) for combining or negating conditions.
- ✓ **Bitwise Operators** (&, |, ^, ~, <<, >>) for manipulating data at the bit level.
- ✓ **Assignment Operators** (=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=) for assigning and updating values.
- ✓ **Unary Operators** (+, -, ++, --, !, ~) for operations on a single operand.
- ✓ **Ternary Operator** (?:) for conditional decisions.
- ✓ **Special Operators** (sizeof, &, *, ->, .) for size checking, memory access, and structure manipulation.



Application of learning 2.1.

Ann is a programmer with career, she is going to hire another to help him, the following is the written exam to pass before to be hired. Declare and initialize the following variables with appropriate data types and values.

1. An integer variable year with a value of 2024.
2. A float variable pi with a value of 3.14.
3. A double variable gravity with a value of 9.81.
4. A character variable grade with a value of 'A'.
5. A string variable greeting with the value "Hello, World!"

Indicative content 2.2: Description of C program structure

Duration:10 hrs

Theoretical Activity 2.2.1: Description of Pre-processor Commands

Tasks:

1: Read carefully and answer the following questions:

i. What is the purpose of pre-processor directives in C?
ii. How is the #include directive used to include header files in C code?
iii. What is the difference between including header files with angle brackets (< >) and double quotes (" ")?
iv. How do you define a constant using the #define directive, and what are the advantages of doing so?
v. What is conditional compilation, and how is it achieved using pre-processor directives in C?

2: Discussion and write the findings on papers.

3: Present the findings to the whole class.

5: Address any questions or concerns.

Key readings 2.2.1. Description of Pre-processor Commands

Preprocessor commands in c

Preprocessor commands, also known as preprocessor directives, are a set of commands used in programming to control the behavior of the preprocessor, a component of the compiler. Preprocessor commands are typically used in languages like C, C++, and other languages that support the C preprocessor. These directives are processed before the actual compilation of the code and are used to perform tasks such as including header files, conditional compilation, and defining macros. Here's a description of some common preprocessor commands:

#include: This directive is used to include the contents of a header file in your source code. It is used to make external code or libraries accessible to your program. For **example:**

#include <stdio.h>

#define: This directive is used to create macros, which are symbolic names for values or code fragments. Macros are replaced by their definitions during preprocessing. For **example:**

#define PI 3.14159265

#ifdef, #ifndef, #else, and #endif: These directives are used for conditional compilation. They allow you to include or exclude parts of your code based on preprocessor-defined conditions. For example:

```
#ifdef DEBUG
// Debugging code here
#else
// Release code here
#endif
```

#undef: This directive is used to undefine a previously defined macro. It removes the macro definition. For example:

```
#define DEBUG_MODE
// ...
#undef DEBUG_MODE
```

#ifdef, #ifndef, #else, and #endif: These directives are used for conditional compilation. They allow you to include or exclude parts of your code based on preprocessor-defined conditions. For example:

```
#ifdef DEBUG
// Debugging code here
#else
// Release code here
#endif
```

#error: This directive is used to generate a compilation error with a custom error message. It can be helpful for enforcing certain conditions during compilation. For example:

```
#ifndef MY_CONST
#error "MY_CONST is not defined!"
#endif
```

#pragma: The #pragma directive is used to provide additional information to the compiler or control certain compiler-specific behavior. Its usage and effects can vary between different compilers.

#line: This directive allows you to change the line number and filename reported by the compiler. It is often used in code generation tools and for debugging purposes.

#ifdef, #ifndef, #else, and #endif: These directives are used for conditional compilation. They allow you to include or exclude parts of your code based on preprocessor-defined conditions. For example:

```
#ifdef DEBUG
// Debugging code here
#else
// Release code here
#endif
```

These preprocessor commands are used to manipulate the code before the actual compilation process, enabling you to customize the behavior of your program based on various conditions and requirements. They are specific to languages like C and C++ that support the C preprocessor. Other programming languages may have different mechanisms for achieving similar goals

Theoretical Activity 2.2.2: Description of c functions

Tasks:

1: Read carefully and answer the following questions:

- **i.** What is a function in C, and why is it used in programming?
- **ii.** How do you declare a function in C?
- **iii.** Explain the difference between types of function in C.
- **iv.** Give an example of a types function.
- **v.** What is the purpose of a return statement in a C function, and how is it used?

2: Discussion and write the findings on papers.

3: Ask trainees to present their findings to the whole class.

5: Address any questions or concerns.

6: Read the Key readings **2.2.2**

Key readings 2.2.2: Description of c functions

Functions in C

Definition of a Function

A **function** in C is a self-contained block of code designed to perform a specific task. Functions are the building blocks of C programs and help in organizing code, improving readability, and reusing code across the program.

Benefits of Using Functions

- ✓ **Modularity**: Functions allow you to divide a complex program into simpler, smaller, manageable parts.
- ✓ **Reusability**: Functions can be reused across the program and even in other programs.
- ✓ **Maintainability**: Functions make code easier to maintain by isolating specific tasks.
- ✓ **Debugging**: By separating code into functions, errors can be identified and fixed more quickly.

Types of Functions in C

C functions can be broadly categorized into two types:

- **Library Functions**:
- ✓ Predefined functions provided by C standard libraries (e.g., printf(), scanf(), strcpy(), etc.).
- ✓ They are ready to use and save development time.
- ✓ Declared in header files (like stdio.h, math.h).
- **User-defined Functions**:
- ✓ Functions created by the programmer to perform specific tasks.
- ✓ Allow for greater flexibility and program customization.
- ✓ Consist of a function declaration, definition, and call.
- **Structure of a User-defined Function**

A typical user-defined function in C consists of three parts:

1. **Function Declaration (Prototype)**:
- ✓ Provides the compiler with information about the function name, return type, and parameters.
- ✓ Syntax:

return_type function_name(parameter_list);

- ✓ Example: int add(int, int);

2. **Function Definition**:
- ✓ Contains the actual code or statements that define what the function does.
- ✓ Syntax:

```
return_type function_name(parameter_list) {
    // Function body
```

}

- ✓ Example:

```
int add(int a, int b) {
   return a + b;
}
```

3. **Function Call**:

- ✓ The place in the program where the function is called to execute.
- ✓ Syntax: function_name(arguments);
- ✓ Example: int sum = add(5, 3);

Function Parameters

Functions can accept inputs called **parameters**:

- ✓ **Formal Parameters**: Defined in the function declaration and definition (e.g., int a, int b in add(int a, int b)).
- ✓ **Actual Parameters**: The real values or arguments passed to the function when it is called (e.g., 5, 3 in add(5, 3)).

- **Return Type of Functions**

- ✓ **void Return Type**: Indicates that the function does not return a value.
- ✓ **Non-void Return Type**: Specifies the type of value that the function returns (e.g., int, char, float).

- **Return Statement**

The **return statement** is used to:

- ✓ End the function execution.
- ✓ Return a value from the function to the caller, if applicable.

Example:

```
int multiply(int x, int y) {
   return x * y; // Returns the product of x and y
}
```

Scope and Lifetime of Variables in Functions

- ✓ **Scope**: Refers to the part of the program where a variable is accessible. Variables defined inside a function are local to that function.
- ✓ **Lifetime**: Refers to the duration a variable exists in memory. Local variables in functions have automatic storage duration; they exist only during function execution.

- **Recursion**

- ✓ A **recursive function** is a function that calls itself.
- ✓ Useful for solving problems that can be broken down into smaller subproblems, such as factorial calculation, Fibonacci series, etc.

- ✓ Example:

```
int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n - 1);
}
```

- **Examples of Function Usage**

Example 1: Function with No Return Type and No Parameters

```
#include <stdio.h>

void printMessage() {
    printf("Hello, World!\n");
}
int main() {
    printMessage(); // Function call
    return 0;
}
```

Example 2: Function with Return Type and Parameters

```
#include <stdio.h>

int add(int a, int b) {
    return a + b; // Returns the sum of a and b
}

int main() {
    int sum = add(5, 3); // Function call
    printf("Sum is: %d\n", sum); // Output: Sum is: 8
    return 0;
}
```

Theoretical Activity 2.2.3: Description of Variables

Tasks:

1: you are requested to read the following questions and answer it:

i. What is a variable in the context of C programming, and why are they important?

ii. How do you think variables help us store and manipulate data in a C program? Can you provide examples?

iii. Why is it important to choose meaningful names for variables? What could go wrong if you use vague or unclear names?

iv. What are some of the basic data types in C, and how do they affect the types of values a variable can store?

v. When you declare a variable, what information must you specify, and why is it necessary?

vi. Differentiate declaring and defining a variable in C? What's the key distinction?

vii. What happens if you try to use a variable before declaring it? Why is it important to declare a variable before using it?

viii. How do you assign a value to a variable in C programming? Can you provide an example?

ix. Why do you think it's essential to understand the scope of a variable in a program? How can variable scope impact your code?

x. Describe the process of initialising a variable in C, and why is it considered best practices?

2: Discussion and write the findings on paper

3: Present their findings to the whole class.

4: Address any questions or concerns.

5: Read the key reading **2.2.3 Trainee Manual**

Key readings 2.2.3: Description of Variables

Variables in C Programming

In C programming, variables are fundamental components used to store and manipulate data. They are named memory locations where you can store values, and they come in various data types. Here's a description of variables in C:

- **Data Types:**
  C supports various data types, including
  int: Used for storing integer values.
  char: Used for characters, such as letters and symbols.
  float: Used for single-precision floating-point numbers.
  double: Used for double-precision floating-point numbers.
  short: Used for short integers.
  long: Used for long integers.
  unsigned: Used for non-negative integer values.
  void: Used to declare functions that return no value.
  You can also create custom data types using struct, enum, and typedef.

- **Declaration:**

You declare a variable by specifying its data type followed by the variable name. For example:

int age;

char initial;

float price;

Initialization:

Variables can be initialized at the time of declaration by providing an initial value. For example:

int count = 10;

char grade = 'A';

float pi = 3.14159;

Assignment:

You can change the value of a variable using the assignment operator =. For example:

age = 30; // Assigns the value 30 to the variable 'age'.

Scope:

Variables have a scope, which defines where in the code they are accessible. Global variables are declared outside of any function and are accessible from anywhere in the program. Local variables are declared inside a function and are only accessible within that function.

- **Lifetime:**

  The lifetime of a variable refers to the duration it exists in memory. Global variables have a lifetime that spans the entire program's execution, while local variables have a shorter lifetime, limited to the scope of the function in which they are declared.

  Constants:

  In addition to variables, C allows you to define constants using the const keyword. Constants are variables whose values cannot be changed once they are set. For example:

  const int numberOfDaysInAWeek = 7;

- **Dynamic Memory Allocation:**

  C also supports dynamic memory allocation, where you can allocate memory at runtime using functions like malloc() and deallocate it using free(). This is often used to create data structures like arrays and linked lists.

- **Naming Conventions:**

  Variable names in C should follow certain naming conventions, such as starting with a letter, followed by letters, digits, or underscores. They are case-sensitive, meaning myVar and myvar are considered different variables.

- **Data Manipulation:**

Variables are used to perform various operations, such as arithmetic calculations, comparisons, and logical operations. For example:

```
int sum = a + b;
if (age >= 18) {
   // Code for adults
}
```

Variables are essential components of C programming as they allow you to store, modify, and work with data. Understanding data types, scope, lifetime, and naming conventions is crucial for effectively using variables in your C programs.

Practical Activity 2.2.4: Apply Variables in C programming

Task:

1: Read the following scenario and perform tasks described below:

In your formed group apply the concepts of variables, data types, initialization, and arithmetic operations in C programming by creating a simple program to calculate the areas of a rectangle and a circle.

2: By using the key reading **2.2.4,** apply Variables in C programming

3: write their finding on paper

4: present their findings to the whole class

Key readings 2.2.4 Apply Variables in C programming

1. Applying variables in C

In C programming, you can declare variables by specifying their data type followed by the variable name. Here are some examples of variable declarations in C:

- **Integer Variable:**
  int age;
  This declares an integer variable named age without initializing it.
- **Character Variable:**
  char grade;
  This declares a character variable named grade.
- **Floating-Point Variable:**
  float temperature;
  This declares a floating-point variable named temperature.
- **Double-Precision Floating-Point Variable:**

double pi;
This declares a double-precision floating-point variable named pi.

- **Multiple Variable Declarations:**
  You can declare multiple variables of the same data type on the same line:
  int x, y, z;
  This declares three integer variables named x, y, and z.
- **Initializing Variables:**
  You can also declare and initialize variables in a single line:
  int count = 10;
  char initial = 'A';
  float price = 3.99;
  Here, the variables count, initial, and price are declared and initialized with specific values.
- **Constants:**
  You can declare constants using the const keyword:
  const int numberOfDaysInAWeek = 7;
  This declares a constant integer variable named numberOfDaysInAWeek with the value 7. The const keyword ensures that the value cannot be changed later in the program.
- **Global Variables:**
  Global variables are declared outside of any function and are accessible from anywhere in the program. For example:
  int globalVar = 42; // Global variable
  This declares a global integer variable named globalVar with an initial value of 42.
- **Local Variables:**
  Local variables are declared within a function and are only accessible within that function:

```
void myFunction() {
   int localVar = 100; // Local variable
}
```

  In this example, localVar is a local variable that can only be used within the myFunction function.
- **Dynamic Memory Allocation:**
  You can declare pointers to dynamically allocated memory, typically using the malloc function:

```
int *dynamicArray;
dynamicArray = (int *)malloc(10 * sizeof(int)); // Dynamically allocated integer array
```

Here, dynamicArray is a pointer to an integer array allocated dynamically in memory.

These are examples of variable declarations in C, covering various data types and scenarios. Variables are fundamental for storing and manipulating data in C programming.

Theoretical Activity 2.2.5: Description of the Pointer in C

Tasks:

1: Read carefully and answer the following questions:

- **i.** What is a c pointers and why is it an essential concept in programming?
- **ii.** Enumerate two applications of c pointers as used in computer system architecture
- **iii.** Identify types of c pointer in programming language
- **iv.** How to declare pointer in c programming language?
- **v.** How to read the pointer in computer programming language?
- **vi.** Enumerate two advantages of Pointer in programming language.

2: Discussion and write the findings on papers.

3: Present their findings to the whole class.

4: Address any questions or concerns.

5: Read the Key readings **2.2.5 Trainee Manual**

Key readings 2.2.5.: Description of the Pointer in C

1. C Pointer?

- **Definition**: A **pointer** in C is a variable that stores the memory address of another variable. Instead of holding a data value directly, it "points" to the location where the value is stored.
- **Importance**:

✓ **Efficient Memory Management**: Pointers allow direct access and manipulation of memory locations, making memory management more efficient.

✓ **Dynamic Memory Allocation**: With pointers, programmers can dynamically allocate memory during runtime using functions like malloc and calloc.

✓ **Function Argument Passing**: Pointers enable functions to modify variables by passing memory addresses (pass-by-reference), which allows for more efficient and versatile code.

- ✓ **Array and String Management**: They provide a way to navigate and manipulate arrays, strings, and complex data structures easily.

2. Two Applications of C Pointers in Computer System Architecture

1. **Memory Management**:
   - Pointers are crucial for handling dynamic memory allocation (heap memory) and deallocation using functions like malloc(), free(), and realloc(). This is particularly useful for managing large datasets or arrays whose size is determined during runtime.
2. **Pointer Arithmetic**:
   - In system-level programming (e.g., operating systems), pointer arithmetic is essential for navigating memory addresses directly, such as moving through arrays or data structures at the hardware level (e.g., accessing device memory, buffers).

3. Types of C Pointers

- **Null Pointer**:
- ✓ A pointer that does not point to any valid memory location. It's often used to signify that the pointer is not currently pointing to any data (e.g., int *ptr = NULL;).
- **Void Pointer**:
- ✓ A generic pointer type (void *) that can point to any data type. It cannot be dereferenced directly and needs to be cast to the appropriate type before use.
- **Dangling Pointer**:
- ✓ A pointer that references a memory location that has already been freed. Using a dangling pointer can lead to undefined behavior.
- **Wild Pointer**:
- ✓ A pointer that has not been initialized and points to a random memory location, leading to unpredictable behavior.

4. Declare a Pointer in C Programming Language?

- **Syntax**:

  To declare a pointer, you need to specify the data type of the variable it points to, followed by an asterisk (*) and the pointer's name.

  data_type *pointer_name;
  - Example:

    int *ptr; // Pointer to an integer

    char *ch; // Pointer to a character

    float *fptr; // Pointer to a floating-point number

5. Read the Pointer in C Programming Language?

- **Accessing the Address**:
- ✓ You can retrieve the memory address of a variable using the address-of operator (&).
- ✓ Example:

  int var = 10;

int *ptr = &var; // ptr now holds the address of var

- **Dereferencing the Pointer**:

✓ To access the value stored at the memory address, use the dereference operator (*).

✓ Example:

int value = *ptr; // Dereferences the pointer to get the value of var

6. Advantages of Pointers in Programming Language

- **Efficient Array and String Manipulation**:

✓ Pointers allow for efficient navigation through arrays and strings. Instead of copying large amounts of data, pointers simply move references to the data.

- **Dynamic Data Structures**:

✓ Pointers are fundamental in implementing dynamic data structures like linked lists, trees, and graphs, which require dynamic memory allocation and efficient node traversal.

Practical Activity 2.2.6: Apply pointers in C programming

Task:

1: Read the following scenario and perform tasks described below:

In a computer system, the operating system (OS) is responsible for managing memory resources efficiently. A key challenge is managing dynamic memory allocation for multiple applications running simultaneously. For instance, when a user opens several programs, such as a browser, a text editor, and a media player, the OS must allocate and track memory for each application. This is crucial to prevent conflicts like memory leaks or overwriting important data. Write a c program to manage the memory allocation of the OS.

2: By using the key reading **2.2.6,** perform the above situation apply pointers in C programming

3: write their finding on paper

4: present their findings to the whole class

Key readings 2.2.6 Apply pointers in C programming

Pointers are a powerful feature in C programming, enabling direct memory access and manipulation. Here are key applications of pointers:

1. **Dynamic Memory Allocation**:

✓ Pointers are used to allocate memory dynamically at runtime using functions like malloc(), calloc(), and realloc(). This is crucial for managing memory efficiently in scenarios where the required memory size is unknown during compile time.

✓ Example:

```
int *ptr = (int *)malloc(sizeof(int) * 10);  // Allocates memory for 10 integers
```

2. **Function Arguments (Pass by Reference)**:

✓ Pointers allow functions to modify the value of variables by passing the memory address instead of copying the variable's value. This is known as pass-by-reference.

✓ Example:

```
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

3. **Array and String Manipulation**:

✓ Pointers provide efficient navigation and manipulation of arrays and strings. Instead of copying data, pointers can iterate over elements and access them directly.

✓ Example:

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = arr;  // Points to the first element of the array
```

4. **Implementing Data Structures**:

✓ Pointers are essential for implementing complex data structures like linked lists, trees, and graphs. These structures rely on dynamically allocated memory and efficient node traversal using pointers.

✓ Example: Linked List Node

```
struct Node {
   int data;
   struct Node* next;
};
```

5. **Pointer to Pointer (Multidimensional Arrays)**:

✓ Pointers to pointers are used to handle multidimensional arrays or dynamically allocated arrays of arrays.

✓ Example:

```
int **matrix = (int **)malloc(3 * sizeof(int *));
for (int i = 0; i < 3; i++) {
   matrix[i] = (int *)malloc(3 * sizeof(int));
}
```

6. **Accessing Hardware Resources**:

✓ In system programming, pointers are used to access and manipulate hardware resources directly (e.g., memory-mapped I/O). This is common in embedded systems and operating system development.

How can you swap the values of two variables using pointers in C?

Answer: To swap the values of two variables using pointers, you pass the memory addresses of the variables to a function. Inside the function, you use pointer dereferencing to access and swap their values.

Example:

```
#include <stdio.h>

// Function to swap two values using pointers
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
int main() {
   int x = 10, y = 20;
   printf("Before swap: x = %d, y = %d\n", x, y);

   // Call the swap function and pass addresses of x and y
   swap(&x, &y);

   printf("After swap: x = %d, y = %d\n", x, y);
   return 0;
}
```

Output:

```
Before swap: x = 10, y = 20
After swap: x = 20, y = 10
```

Question 2:

How do you dynamically allocate memory for an array using pointers in C?

Answer: In C, you can use the malloc() function to dynamically allocate memory for an array. You assign the result of malloc() to a pointer, which points to the first element of the array.

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int *arr;
   int n = 5;
   // Dynamically allocate memory for an array of 5 integers
   arr = (int *)malloc(n * sizeof(int));
   // Check if memory allocation was successful
   if (arr == NULL) {
      printf("Memory allocation failed!\n");
      return 1;
   }
   // Assign values and print the array
   for (int i = 0; i < n; i++) {
      arr[i] = i + 1;
      printf("%d ", arr[i]);
   }
   // Free the allocated memory
   free(arr);
   return 0;
}
```

Output:

1 2 3 4 5

Points to Remember

Pointers in C are variables that store the memory address of another variable. They can point to various data types such as int, char, arrays, functions, or even other pointers.

Types of Pointers:

- **Null Pointer:** A pointer assigned a null value (0). Useful when no address is assigned initially.
- **Void Pointer:** A generic pointer without a specific data type. It can hold the address of any type and be cast to any type.
- **Wild Pointer:** An uninitialized pointer that points to a random memory location, potentially causing program crashes.

Declaring a Pointer:

- Use the * symbol to declare a pointer, e.g., int *a; for a pointer to an integer.
- Complex pointer declarations, like int (*p)[10], involve precedence rules between *, [], and (), with associativity being left-to-right for [] and () and right-to-left for *.

Example Pointer:

int (*p)(int (*)[2], int (*)void)) is a pointer to a function that takes two parameters:

- A pointer to a one-dimensional array of integers of size two.
- A pointer to a function that takes no arguments and returns an integer.

Pointers are crucial in computer system architecture, enabling direct manipulation of memory addresses for efficient memory management, hardware interaction, and system performance optimization. Key applications include:

- **Dynamic Memory Management**: Pointers manage memory allocation and deallocation at runtime, ensuring optimal use of memory and preventing leaks.
- **Data Structures**: Pointers link nodes in structures like linked lists and trees, enabling flexible, non-contiguous storage.
- **Hardware Interfacing**: Pointers directly access hardware registers through memory-mapped I/O, allowing communication with devices like hard drives and GPUs.
- **Function Pointers and Callbacks**: Pointers reference functions for dynamic execution and handling of system events, useful in drivers and kernel modules.
- **Embedded Systems**: Pointers allow precise memory addressing, essential for resource-constrained systems like real-time controllers and sensor operations.
- using pointers and malloc (e.g., int *dynamicArray = (int *)malloc(10 * sizeof(int));).

Theoretical Activity 2.2.5: Description of Statements & Expressions

Tasks:

1: Read carefully and answer the following questions:

i. What is the fundamental difference between a statement and an expression in C programming, and can you provide examples of each?
ii. How are statements used to control the flow of execution in a C program? Can you give an example of a control statement and its purpose?
iii. Explain the role of expressions in C programming. How are expressions evaluated, and what are some common examples of expressions?

2: Discussion and write the findings on papers.

3: Present their findings to the whole class.

4: Address any questions or concerns.

5: Read the Key readings **2.2.5**

Key readings 2.2.5. Description of Statements & Expressions

Statements & Expressions

In computer programming, both statements and expressions are fundamental concepts. They serve different purposes and have distinct characteristics.

Statements:

A statement is a complete line of code that performs a specific action or task. It represents a command or an instruction to the computer.

Termination: In most programming languages, statements are usually terminated by a semicolon (;). This semicolon marks the end of the statement.

- **Examples of Statements:**

Assignment Statements: x = 10;

Control Flow Statements (if, for, while, switch, etc.):

```
if (condition) {
   // code
}
```

Function Call Statements: printf("Hello, World");

Side Effects: Statements often have side effects, meaning they can modify the state of the program, such as changing variable values, performing I/O operations, or altering control flow.

Return Value: Statements do not have a return value. They are executed for their side effects rather than producing a value.

- **Expressions:**

An expression is a combination of variables, constants, operators, and function calls that produces a single value. Expressions are evaluated to yield a result.

No Semicolon: Expressions do not end with a semicolon; they are evaluated to produce a value, and that value can be used within statements.

- **Examples of Expressions:**

Arithmetic Expressions: x + y

Logical Expressions: a && b

Function Call Expressions: sqrt(25)

Assignment Expressions (Right-hand side): z = x + y; (The right side is an expression.)

No Side Effects: Expressions are typically side-effect-free. They are meant to be evaluated to produce a result without altering the program's state. In functional programming, this property is emphasized.

Return Value: Expressions always have a return value, and this value has a specific data type. For example, an arithmetic expression may return an integer or a floating-point value, while a logical expression returns a Boolean value.

- **Relationship between Statements and Expressions:**
  Statements often contain expressions. For example, an assignment statement typically involves an expression on the right side that calculates the value to be assigned to the variable on the left side.
  int x = 5 + 3; // Here, "5 + 3" is an expression within the assignment statement.
  Expressions can be used within statements to compute values or make decisions based on those values. For example, an "if" statement often relies on the result of a logical expression.

```
if (x > 10) {
   // This is a statement that depends on the expression "x > 10."
}
```

  In summary, statements are instructions or commands that perform actions in a program and can have side effects, while expressions are combinations of values and operators that produce results without side effects and always have a specific data type and a return value. Both statements and expressions are essential in programming and are used in different contexts within a program.

Theoretical Activity 2.2.6: Description of Comments

Tasks:

1: Read carefully and answer the following questions:

i. What are comments in C programming, and what is their primary purpose when writing code?
ii. Explain the different types of comments used in C programming, such as single-line comments and multi-line comments. Can you provide examples of each?

2: Discussion and a write the findings on paper
3: resent their findings to the whole class.
4: Address any questions or concerns.
5: Read the key reading **2.2.6**

Key readings 2.2.6. Description of Comments

Comments in c Programming

In C programming, comments are non-executable parts of the code that provide explanations, clarifications, or documentation to help programmers understand the code. Comments are ignored by the compiler, so they do not affect the execution of the program. There are two types of comments in C: single-line comments and multi-line comments.

Types of Comments in C

1. **Single-Line Comments**
   - **Syntax**: Use two forward slashes (//) to create a single-line comment.
   - **Usage**: Everything following // on that line is considered a comment and will be ignored by the compiler.

   Example:

   ```
   int x = 10;  // This is a single-line comment explaining the variable 'x'
   ```
2. **Multi-Line Comments**
   - **Syntax**: Enclose the comment text between /* and */.
   - **Usage**: Useful for longer comments that span multiple lines. Everything between /* and */ is ignored by the compiler.

   Example:

   ```
   /*
    This is a multi-line comment.
    It can span multiple lines and is useful for detailed explanations or documentation.
   */
   int y = 20;
   ```

Uses of Comments

- **Code Explanation**: Provide explanations for complex or non-intuitive code sections, making it easier for others (or your future self) to understand what the code does.

  ```
  // Calculate the area of a circle
  double area = PI * radius * radius;
  ```
- **Documentation**: Describe the purpose and usage of functions, variables, and data structures within the code.

  ```
  /*
   * Function: calculateSum
   * ----------------------
   * Computes the sum of two integers.
   *
   * a: the first integer
  ```

```
 * b: the second integer
 *
 * returns: the sum of a and b
 */
int calculateSum(int a, int b) {
   return a + b;
}
```

- **Code Organization**: Separate different sections of code for better readability.

```
// Initialize variables
int a = 5;
int b = 10;

// Perform calculations
int sum = a + b;
```

- **Debugging and Development**: Temporarily disable parts of the code by commenting them out during development or debugging.

```
// printf("This line is commented out and won't execute\n");
```

Best Practices for Using Comments

- **Be Clear and Concise**: Write comments that are clear and to the point. Avoid unnecessary comments that state the obvious.

```
// Increment counter
counter++;  // This is clear and does not require additional comments
```

- **Keep Comments Updated**: Ensure that comments are updated when code changes. Outdated comments can be misleading.
- **Use Comments Sparingly**: Rely on well-written and self-explanatory code as much as possible. Use comments to clarify only when necessary.
- **Avoid Redundancy**: Don't state what the code is doing if it's already clear from the code itself. Instead, explain why the code is doing something.

```
// Bad Comment
int count = 0; // Initialize count to zero

// Good Comment
int count = 0; // Initialize count to track the number of items processed
```

Comments play a crucial role in code readability and maintainability. They help programmers understand, document, and communicate the intent and logic of the code efficiently.

Points to Remember

Preprocessor commands in c

Preprocessor commands, also known as preprocessor directives, are used in programming languages like C and C++ to control the behavior of the preprocessor, which is a component of the compiler. Common preprocessor commands include:

- **#include**: Used to include the contents of a header file into the source code, allowing access to external code or libraries (e.g., #include <stdio.h>).
- **#define**: Used to create macros, which are symbolic names for values or code fragments that are replaced by their definitions during preprocessing (e.g., #define PI 3.14159265).
- **#ifdef, #ifndef, #else, #endif**: Used for conditional compilation, allowing code to be included or excluded based on preprocessor-defined conditions (e.g., #ifdef DEBUG).
- **#undef**: Used to undefine a previously defined macro (e.g., #undef DEBUG_MODE).
- **#error**: Generates a compilation error with a custom message, useful for enforcing certain conditions during compilation (e.g., #error "MY_CONST is not defined!").
- **#pragma**: Provides additional information to the compiler or controls certain compiler-specific behavior. Its usage can vary between different compilers.
- **#line**: Changes the line number and filename reported by the compiler, often used in code generation tools and for debugging.

Functions in C

Definition of a Function

A function in C is a block of code that performs a specific task, helping to organize, improve readability, and reuse code.

Types of Functions in C

1. **Library Functions**: Predefined in C standard libraries (e.g., printf(), scanf()).
2. **User-defined Functions**: Created by the programmer to perform custom tasks.

Variables in C Programming

Variables are fundamental components in C used to store and manipulate data. They represent named memory locations for storing values and come in various data types.

- **Data Types**

C supports several basic data types:

- ✓ **int**: Stores integer values.
- ✓ **char**: Stores characters, such as letters and symbols.
- ✓ **float**: Stores single-precision floating-point numbers.
- ✓ **double**: Stores double-precision floating-point numbers.
- ✓ **short**: Stores short integers.
- ✓ **long**: Stores long integers.
- ✓ **unsigned**: Stores non-negative integer values.

- ✓ **void**: Used for functions that return no value.

Application of variables

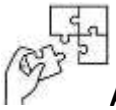
- **Variable Declarations**: Variables in C are declared by specifying a data type followed by the variable name (e.g., int age;, char grade;).
- **Multiple Declarations**: Multiple variables of the same type can be declared on a single line (e.g., int x, y, z;).
- **Initialization**: Variables can be declared and initialized in a single line (e.g., int count = 10;, char initial = 'A';).
- **Constants**: Use the const keyword to declare constants whose values cannot be changed (e.g., const int numberOfDaysInAWeek = 7;).
- **Global Variables**: Variables declared outside any function are global and accessible throughout the program (e.g., int globalVar = 42;).
- **Local Variables**: Variables declared within a function are local and accessible only within that function (e.g., int localVar = 100;).
- **Dynamic Memory Allocation**: Dynamically allocate memory using pointers and malloc (e.g., int *dynamicArray = (int *)malloc(10 * sizeof(int));).

Statements & Expressions

- **Statements:** A statement is a complete line of code that performs an action, typically ending with a semicolon and does not return a value but may have side effects.
- **Expressions:** An expression combines variables, constants, operators, and function calls to produce a single value and always returns a value but does not cause side effects.
- **Statement Examples:** Includes assignment (e.g., x = 10;), control flow (e.g., if (condition) {...}), and function calls (e.g., printf("Hello, World");).
- **Expression Examples:** Includes arithmetic (e.g., x + y), logical (e.g., a && b), and function call expressions (e.g., sqrt(25)).
- **Relationship:** Statements often contain expressions to calculate values or make decisions, such as in assignment or control flow statements.

Comments in c programming

- **Types of Comments:** In C, comments can be single-line (using //) or multi-line (enclosed in /* */).
- **Code Explanation:** Use comments to clarify complex or non-intuitive code, making it easier to understand.
- **Documentation:** Comments can describe the purpose, usage, and behavior of functions, variables, and data structures.
- **Code Organization:** Comments can help separate and organize different sections of code for better readability.
- **Best Practices:** Comments should be clear, concise, up-to-date, used sparingly, and focus on explaining the "why" rather than the "what" of the code.



Application of learning 2.2.

Rwanda BGD want to manage your daily expenses. As you are a programmer, declare two integer variables representing amounts spent on two different items. Perform addition, subtraction, multiplication, and division on these amounts to understand your total and average spending. Print the results to see your calculations.

Indicative content 2.3: Applications of Condition Statements

Duration: 10hrs

Theoretical Activity 2.3.1: Description of the conditional statement in C

Tasks:

1: Read carefully and respond to the following questions:

I. What is a conditional statement, and why is it an essential concept in programming?

II. Provide an example of a real-life situation where you would use a conditional statement to make a decision or perform different actions based on a specific condition.

III. Explain the 'if' statement in C programming, including how it works, its syntax, and its purpose? Additionally, clarify the concept of 'else if' statements and their use cases compared to multiple 'if' statements, and describe when and how to use nested 'if' statements in your code."

IV. What is a "switch" statement, and how does it differ from "if" statements?

V. Describe the conditions or scenarios in which you would choose to use "if" statements over "switch" statements, and vice versa.

2: Discussion and write the findings on paper

3: Present their findings to the whole class.

4: Address any questions or concerns.

5: Read the key reading **2.3.1**

Key readings 2.3.1: Description of the conditional statement in C

Conditional statements in c programming

In C programming, conditional statements are used to make decisions and execute code based on certain conditions or criteria. Conditional statements allow you to control the flow of a program, enabling it to perform different actions depending on whether specific conditions are met. There are primarily three types of conditional statements in C: if, else if, and else.

- **if Statement:**

  The if statement is used to test a single condition. If the condition is true, the code inside the if block is executed. If the condition is false, the code inside the if block is skipped.

Syntax:

```
if (condition) {
   // Code to execute when the condition is true
}
```

- **else if Statement:**

  The else if statement allows you to test multiple conditions in sequence. It is used when you have multiple conditions, and you want to execute different code blocks based on which condition is true.

  Syntax:

```
if (condition1) {
   // Code to execute when condition1 is true
} else if (condition2) {
   // Code to execute when condition2 is true
} else {
   // Code to execute if none of the conditions are true
}
```

- **else Statement:**

  The else statement is used in conjunction with the if statement and provides a code block to execute when the if condition is false.

  Syntax:

```
if (condition) {
   // Code to execute when the condition is true
} else {
   // Code to execute when the condition is false
}
```

  Example:

  Here is a simple example of an if statement in C:

```
#include <stdio.h>
int main() {
   int num = 10;

   if (num > 5) {
      printf("The number is greater than 5.\n");
   }
   return 0;
}
```

  In this example, if the condition num > 5 is true, the message "The number is greater than 5." is printed to the console.

- **Notes:**
  Conditions in conditional statements are expressions that evaluate to either true or false (1 or 0). You can use comparison operators (e.g., >, <, ==, !=, etc.) to build conditions.
  Logical operators (&&, ||, !) can be used to combine multiple conditions. The else if and else parts are optional, depending on the complexity of your decision-making logic.
  You can nest conditional statements within one another to create more complex decision structures.
  Conditional statements are essential for controlling the program's flow and enabling it to make decisions, which is a fundamental aspect of programming. They allow you to create flexible and responsive code that can adapt to different situations.

Practical Activity 2.3.2: Solving problem using if statement in c programming

Task:

1: Read carefully the following scenario and perform the tasks described below:

You need to implement a feature in a shopping application that checks if a customer is eligible for a discount based on the amount they spend. The rules for the discount are as follows:

- If the total purchase amount is greater than or equal to $100, the customer receives a 10% discount.
- If the total purchase amount is less than $100, no discount is applied.

2: By using **reading 2.3.2,** Discuss by writing the pseudocode of a given task.

3: convert the written pseudocode into program code by using if statement syntax.

4: present their finding to the class.

5: Ask trainees to ask clarification where necessary.

Key readings 2.3.2: Solving problem using if statement in c programming

If statement in C programming

An if statement in C programming is a control structure used to execute a block of code only if a specified condition is true. It is fundamental to decision-making in

programming, allowing the program to take different actions based on varying conditions.

Syntax of if Statement

```
if (condition) {
   // code to execute if condition is true
}
```

Components

1. **Condition**:

- This is a logical expression that evaluates to either true (non-zero) or false (zero). The condition is placed within parentheses ().

2. **Block of Code**:

- This block is enclosed within curly braces {} and contains the statements that should be executed if the condition is true.
- If the block contains only one statement, the braces can be omitted, though it's generally good practice to include them for clarity and to prevent errors when modifying the code later.

How It Works

- **Evaluation**: The condition inside the if statement is evaluated first.
- **Execution**: If the condition is true (non-zero), the block of code within the if statement is executed. If the condition is false (zero), the program skips the block and continues with the next statement after the if block.

Example

```
#include <stdio.h>
int main() {
   int number;

   // Prompt the user to enter a number
   printf("Enter a number: ");
   scanf("%d", &number);

   // Check if the number is positive
   if (number > 0) {
      printf("The number is positive.\n");
   }

   // Continue with the rest of the program
   printf("Program continues...\n");

   return 0;
}
```

Practical Activity 2.3.3: Solving problem using else if statement in c

Task:

1: Read the following scenario and by using key readings 2.3.3, perform the tasks described below:

You need to implement a feature in an educational application that assigns a letter grade to a student based on their score. The grading criteria are as follows:

- A score of 90 or above receives an "A".
- A score of 80 to 89 receives a "B".
- A score of 70 to 79 receives a "C".
- A score of 60 to 69 receives a "D".
- A score below 60 receives an "F".

2: By using **Reading 2.3.3.,** Discuss by writing the pseudocode of a given task.

3: convert the written pseudocode into program code by using else if statement syntax.

4: Present their finding to the class.

5: Ask clarification where necessary.

Key readings 2.3.3 Solving problem using else if statement in c

else if statement in c

The else if statement in C is used to evaluate multiple conditions in sequence. It extends the basic if statement to allow for more complex decision-making processes by checking additional conditions if the initial if condition evaluates to false. This helps create more nuanced and detailed control flows within the program.

Syntax

```
if (condition1) {
   // Code to be executed if condition1 is true
} else if (condition2) {
   // Code to be executed if condition2 is true
} else if (condition3) {
   // Code to be executed if condition3 is true
} else {
   // Code to be executed if none of the above conditions are true
```

}

Explanation

- **if Block**: Evaluates condition1. If condition1 is true, the corresponding block of code executes, and the rest of the else if and else blocks are skipped.
- **else if Blocks**: Each subsequent else if evaluates its condition (condition2, condition3, etc.) if all previous conditions were false. The first true condition's block of code executes.
- **else Block**: Executes if none of the preceding conditions are true. This is optional but provides a fallback for when none of the conditions are met.

Practical Activity 2.3.4: Solving problem using switch statement in c

Task:

1: Read the following situation and perform the tasks described below:

You want to create a simple calculator application that allows users to choose an arithmetic operation (addition, subtraction, multiplication, or division) from a menu. The program should then perform the selected operation on two numbers provided by the user.

2: By using key **reading 2.3.4** Discuss by writing the pseudocode of a given task.

3: convert the written pseudocode into program code by using switch statement syntax.

4: Present their finding to the class.

5: Ask clarification where necessary.

Key readings 2.3.4: Solving problem using switch statement in c

Switch statement

The switch statement in C is a control flow statement that allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case. The switch statement provides a way to execute different parts of code based on the value of an expression, offering a clearer and more structured alternative to multiple if-else if-else statements when dealing with numerous conditions.

Syntax

```
switch (expression) {
   case value1:
      // Code to be executed if expression equals value1
```

```
        break;
    case value2:
        // Code to be executed if expression equals value2
        break;
    ...
    default:
        // Code to be executed if expression doesn't match any case
}
```

Explanation

- **switch (expression)**: The expression is evaluated once, and its value is compared with the values of each case label.
- **case value:**: If the expression matches value, the corresponding block of code is executed. The break statement exits the switch block.
- **default:**: The default case is optional. It executes if none of the case values match the expression. It acts as a fallback.

Practical Activity 2.3.5: Solving problem using nested conditions in c

Task:

1: Read carefully the following situation and perform the tasks described below:
You need to decide what to wear based on the weather conditions and the type of event you plan to attend. The choices depend on whether it is a formal event or a casual outing, combined with the weather being sunny, rainy, or cold.
2: Use key reading **2.3.5**, and write the pseudocode of a given task.
3: convert the written pseudocode into program code by using nested if syntax.
4: Present their finding to the class.
5: Ask clarification where necessary.

Key readings 2.3.5: Solving problem using nested conditions in c

Nested conditions in C involve placing one or more if, else if, or else statements inside another if, else if, or else block. This allows for more complex decision-making processes by enabling the evaluation of multiple, layered conditions.

Syntax

```
if (condition1) {
    // Code to be executed if condition1 is true
```

```
        if (condition2) {
            // Code to be executed if condition1 and condition2 are true
        } else {
            // Code to be executed if condition1 is true and condition2 is false
        }
    } else {
        // Code to be executed if condition1 is false
        if (condition3) {
            // Code to be executed if condition1 is false and condition3 is true
        } else {
            // Code to be executed if condition1 and condition3 are false
        }
    }
```

Explanation

- **Primary if Condition**: Evaluates condition1. If true, the corresponding block executes, otherwise the else block (if present) executes.
- **Nested if Condition**: Inside the primary if or else block, additional if, else if, or else statements can be nested to evaluate further conditions.
- **Multiple Layers**: Nested conditions can be multiple levels deep, allowing for intricate and detailed decision-making logic.

Points to Remember

Conditional statement in c programming

Essential for decision-making in code, allowing different actions based on specific conditions.

1. **'if' Statement in C:**
    - Used to execute code blocks only if a condition is true. The syntax includes if (condition) { code }.
    - else if statements allow checking multiple conditions, providing more control than multiple standalone if statements.
    - Nested if statements enable more complex decision-making by placing if statements inside other if statements.
2. **Switch Statements:**
    - Used for handling multiple possible values of a single variable, often simpler and more readable than using multiple if statements.

If statement in C programming

An if statement in C programming is a control structure used to execute a block of code only if a specified condition is true.

Syntax of if Statement

```
if (condition) {
   // code to execute if condition is true
}
```

Components

- **Condition**
- **Block of Code**

else if statement in c

The else if statement in C programming is used to handle multiple conditional branches in a decision-making process

Syntax of else if

```
if (condition1) {
   // Code to execute if condition1 is true
} else if (condition2) {
   // Code to execute if condition1 is false and condition2 is true
} else if (condition3) {
   // Code to execute if condition1 and condition2 are false and condition3 is true
} else {
   // Code to execute if none of the above conditions are true
}
```

Components

- **Initial if Statement**
- **else if Statements**
- **Optional else Statement**

Switch Statement

Executes one code block among many options based on a variable's value, offering a cleaner alternative to multiple if...else if statements.

- **Expression Evaluation:**
  The switch statement evaluates an expression that must be an integer or character, and this value is compared against case labels.
- **Case Labels:**
  Case labels represent possible values for the expression. If a match is found, the associated code block is executed.
- **Break Statement:**
  Used to exit the switch block after a case is executed; prevents "fall-through" to subsequent cases unless intended.
- **Default Case:**
  Provides a fallback option if no case matches; optional but recommended to handle unexpected values.

Nested conditions in C

Nested conditions in C refer to using conditional statements, such as if, else if, and else, within other conditional statements. This structure allows you to create complex decision-making processes by evaluating multiple criteria in a hierarchical manner.

```
if (condition1) {
   // Code block executed if condition1 is true
   if (condition2) {
      // Code block executed if condition1 and condition2 are true
   } else {
      // Code block executed if condition1 is true and condition2 is false
   }
} else {
   // Code block executed if condition1 is false
   if (condition3) {
      // Code block executed if condition1 is false and condition3 is true
   } else {
      // Code block executed if condition1 and condition3 are false
   }}
```

Application of learning 2.3.

ABC Company seller of movie. They need to hire a programmer to Write a program that calculates the price of a movie ticket based on the customer's age and the time of day they wish to watch the movie. The program should apply different pricing rules based on these conditions.

Indicative content 2.4: Applications of Loops

Duration: 10hrs

Theoretical Activity 2.4.1: Description of Loops

Tasks:

1: Read carefully and respond to the following questions:

I. What is a loop in programming, and why do we use loops?
II. How does a "while" loop work, and what's its primary purpose?
III. What are the key differences between a "for" loop and a "while" loop?
IV. Can you provide a practical example of when you might use a loop in a real-life situation?
V. Why is it important to have a clear exit condition for loops to prevent them from running infinitely?
VI. How does a loop help in avoiding repetitive and redundant code in a program?

2: Discussion and write the findings on paper

3: Present their findings to the whole class.

4: Ask questions or concerns.

5: orients them to read the key reading **2.4.1**

Key readings 2.4.1. Description of Loops

Loops in c programming

Loops in C programming allow a set of instructions to be executed repeatedly, either for a specific number of times or until a certain condition is met. They are fundamental for performing repetitive tasks efficiently and reducing code redundancy.

Types of Loops in C

1. **for Loop**
2. **while Loop**
3. **do-while Loop**

1. for Loop

The for loop is used when the number of iterations is known beforehand. It combines initialization, condition checking, and increment/decrement in one line.

Syntax

```
for (initialization; condition; increment/decrement) {
   // Code to be executed
}
```

Example

```
#include <stdio.h>

int main() {
   for (int i = 1; i <= 5; i++) {
      printf("%d\n", i);
   }
   return 0;
}
```

Explanation

- **Initialization**: Sets the starting value of the loop control variable (e.g., int i = 1).
- **Condition**: Checked before each iteration; if true, the loop body executes (e.g., i <= 5).
- **Increment/Decrement**: Updates the loop control variable after each iteration (e.g., i++).

2. while Loop

The while loop is used when the number of iterations is not known in advance. It continues executing as long as the given condition is true.

Syntax

```
while (condition) {
   // Code to be executed
}
```

Example

```
#include <stdio.h>

int main() {
   int i = 1;
   while (i <= 5) {
      printf("%d\n", i);
      i++;
   }
   return 0;
}
```

Explanation

- **Condition**: Checked before each iteration. If true, the loop body executes; otherwise, the loop terminates.

- **Body Execution**: The loop body can update the loop control variable to eventually make the condition false and exit the loop.

3. do-while Loop

The do-while loop is similar to the while loop but guarantees that the loop body is executed at least once. The condition is checked after executing the loop body.

Syntax

```
do {
   // Code to be executed
} while (condition);
```

Example

```
#include <stdio.h>

int main() {
   int i = 1;
   do {
      printf("%d\n", i);
      i++;
   } while (i <= 5);
   return 0;
}
```

Explanation

- **Initial Execution**: Executes the loop body once before checking the condition.
- **Condition**: Checked after the loop body execution to determine if the loop should continue.

Loop Control Statements

C provides several statements to control the flow of loops:

- **break**: Exits the loop immediately.
- **continue**: Skips the remaining code in the current iteration and proceeds to the next iteration.
- **goto**: Transfers control to a labeled statement (use with caution as it can lead to unstructured code).

Example with break and continue

```
#include <stdio.h>

int main() {
   for (int i = 1; i <= 10; i++) {
      if (i == 5) {
         continue; // Skip the current iteration when i is 5
      }
```

```
        if (i == 8) {
            break; // Exit the loop when i is 8
        }
        printf("%d\n", i);
    }
    return 0;
}
```

Nested Loops in C

Nested loops in C are loops within loops. This means placing one loop inside the body of another loop. They are particularly useful for performing operations that require multiple levels of iteration, such as working with multi-dimensional arrays or generating complex patterns.

Syntax

```
for (initialization; condition; increment/decrement) {
    for (initialization; condition; increment/decrement) {
        // Code to be executed in the inner loop
    }
    // Code to be executed in the outer loop
}
```

Explanation

- **Outer Loop**: The outer loop executes its body a certain number of times.
- **Inner Loop**: For each iteration of the outer loop, the inner loop executes its body completely.

Key Points

- **Efficiency**: Loops reduce code redundancy by allowing repeated execution with minimal syntax.
- **Versatility**: Different loop types allow handling both fixed and variable iteration scenarios.
- **Control Flow**: Loop control statements (break, continue) provide finer control over loop execution.

Advantages

- **Reduces Code Redundancy**: Loops minimize the need for repeating code.
- **Enables Complex Iterations**: Handle repetitive tasks efficiently.
- **Flexible and Powerful**: Can iterate over arrays, collections, and implement complex algorithms.

Practical Activity 2.4.2: Solving problem using for loop in c programming

Task:

1: Read carefully the given scenario and perform tasks described below:

You are developing a simple voting system where users can vote for different candidates. After collecting the votes, you need to count and display the number of votes each candidate received.

2: By using **Key readings 2.4.2,** Discuss by writing the pseudocode of a given task.

3: Convert the written pseudocode into program code by using for loop syntax.

4: Present their finding to the class.

5: Ask clarification where necessary.

Key readings 2.4.2: Solving problem using for loop in c programming

The for loop in C is a control flow statement that allows code to be executed repeatedly based on a given condition. It is typically used when the number of iterations is known beforehand. The for loop combines initialization, condition checking, and increment/decrement in a single line, making it concise and easy to read.

Syntax

```
for (initialization; condition; increment/decrement) {
   // Code to be executed
}
```

Components

1. **Initialization**: This part is executed once at the beginning of the loop. It is usually used to initialize the loop control variable.
2. **Condition**: This part is evaluated before each iteration of the loop. If the condition is true, the loop body is executed. If it is false, the loop terminates.
3. **Increment/Decrement**: This part is executed after each iteration of the loop body. It is usually used to update the loop control variable.

Flowchart

1. **Initialization**: Set the starting value of the loop control variable.
2. **Condition**: Check if the condition is true.
   - **True**: Execute the loop body.
   - **False**: Exit the loop.
3. **Increment/Decrement**: Update the loop control variable.
4. Repeat steps 2 and 3 until the condition is false.

Practical Activity 2.4.3: Solving problem using while loop in c programming

Task:

1: Read carefully the situation to perform tasks described below:

You are developing a basic ATM system that allows users to withdraw money from their account. The system needs to repeatedly prompt the user to enter the amount they wish to withdraw, ensure the amount does not exceed their balance, and check if it is a valid withdrawal amount. The loop should continue until the user successfully withdraws the amount or decides to cancel the transaction.

2: By using **Key readings 2.4.3.**, discuss by writing the pseudocode of a given task.

3: convert the written pseudocode into program code by using while syntax.

4: present their finding to the class.

5: Ask clarification where necessary.

Key readings 2.4.3: Solving problem using while loop in c programming

While loop in c programming

The while loop in C is a control flow statement that allows code to be executed repeatedly based on a given condition. It is typically used when the number of iterations is not known beforehand and depends on a condition that is evaluated before each iteration.

Syntax

```
while (condition) {
   // Code to be executed
}
```

Components

1. **Condition**: The expression evaluated before each iteration of the loop. If the condition is true, the loop body is executed. If it is false, the loop terminates.
2. **Loop Body**: The code block that is executed as long as the condition is true.

Flowchart

1. **Condition**: Check if the condition is true.
   - **True**: Execute the loop body.
   - **False**: Exit the loop.
2. **Loop Body**: Execute the code inside the loop.
3. Repeat steps 1 and 2 until the condition is false.

Example

Consider a program that prints numbers from 1 to 5:

```
#include <stdio.h>

int main() {
   int i = 1; // Initialization

   while (i <= 5) { // Condition
      printf("%d\n", i);
      i++; // Increment
   }

   return 0;
}
```

Output

1
2
3
4
5

Explanation

- **Initialization**: int i = 1 sets the starting value of i to 1.
- **Condition**: i <= 5 checks if i is less than or equal to 5. If true, the loop body executes.
- **Increment**: i++ increments i by 1 after each iteration, ensuring that i eventually exceeds 5 and the loop terminates.

Practical Activity 2.4.4: Solving problem using do while in c programming

Task:

1: Read carefully the situation and perform tasks described below:

You are developing a user authentication system that prompts the user to enter their password. The system should continue to prompt the user until they enter the correct password or choose to cancel after a certain number of failed attempts. You want to ensure that the prompt is displayed at least once, regardless of whether the password is correct on the first try.

2: By using Reading **2.4.4:** Discuss by writing the pseudocode of a given task.

3: convert the written pseudocode into program code by using do while syntax.

4: Present their finding to the class.

5: Ask clarification where necessary.

Key readings 2.4.4 Solving problem using do while in c programming

do-while loop

The do-while loop in C is a control flow statement that allows code to be executed repeatedly based on a given condition. Unlike the while loop, the do-while loop guarantees that the loop body is executed at least once before the condition is tested.

Syntax

```
do {
   // Code to be executed
} while (condition);
```

Components

1. **Loop Body**: The code block that is executed at least once and then repeatedly as long as the condition is true.
2. **Condition**: The expression evaluated after each iteration of the loop. If the condition is true, the loop body is executed again. If it is false, the loop terminates.

Flowchart

1. **Loop Body**: Execute the code inside the loop.
2. **Condition**: Check if the condition is true.
   - **True**: Execute the loop body again.
   - **False**: Exit the loop.
3. Repeat steps 1 and 2 until the condition is false.

Example

Consider a program that prints numbers from 1 to 5:

```
#include <stdio.h>

int main() {
   int i = 1; // Initialization

   do {
      printf("%d\n", i); // Loop body
      i++; // Increment
   } while (i <= 5); // Condition

   return 0;
}
```

Output

Copy code

```
1
2
```

3
4
5

Explanation

- **Initialization**: int i = 1 sets the starting value of i to 1.
- **Loop Body**: printf("%d\n", i); i++; prints the value of i and then increments i by 1.
- **Condition**: i <= 5 is checked after the loop body executes. If true, the loop body executes again; if false, the loop terminates.

Practical Activity 2.4.5: Solving problem using nested loops in c

Task:

1: Read the following situation then perform the tasks described below:

You are developing a program that needs to generate and display a multiplication table for numbers from 1 to 10. This involves creating a table where each cell in the table represents the product of the row and column numbers.

2: By using the Reading **2.4.5.** Discuss by writing the pseudocode of a given task.

3: Convert the written pseudocode into program code by using nested loop syntax.

4: Present their finding to the class.

5: Ask clarification where necessary.

Key readings 2.4.5: Solving problem using nested loops in c programming

Nested loops in C

Nested loops in C are loops that exist within another loop. They are useful for dealing with multi-dimensional data structures or scenarios requiring multiple levels of iteration. A typical use case for nested loops is working with matrices or generating patterns.

Syntax

The basic structure of nested loops is as follows:

```
for (initialization; condition; increment/decrement) {
   for (initialization; condition; increment/decrement) {
      // Code to be executed
   }
}
```

You can also use while and do-while loops in a nested fashion:

```
while (condition) {
   while (condition) {
      // Code to be executed
   }
}
do {
   do {
      // Code to be executed
   } while (condition);
} while (condition);
```

Example: Multiplication Table

Consider a program that prints a 10x10 multiplication table:

```
#include <stdio.h>

int main() {
   // Outer loop for rows
   for (int i = 1; i <= 10; i++) {
      // Inner loop for columns
      for (int j = 1; j <= 10; j++) {
         printf("%4d", i * j); // Print product with width of 4
      }
      // Newline after each row
      printf("\n");
   }
   return 0;
}
```

Output

```
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

Explanation

- **Outer Loop**: The outer loop (for (int i = 1; i <= 10; i++)) iterates over the rows.
- **Inner Loop**: The inner loop (for (int j = 1; j <= 10; j++)) iterates over the columns.

- **Product Calculation**: i * j computes the product of the current row and column indices.
- **Output Formatting**: printf("%4d", i * j) ensures the products are aligned in a 4-character-wide field.

Points to Remember

Loops in c programming

- **For Loop:**
  Used when the number of iterations is known beforehand; includes initialization, condition, and increment/decrement in one line.
- **While Loop:**
  Ideal for scenarios where the number of iterations is not predetermined; continues as long as the condition is true.
- **Do-While Loop:**
  Similar to a while loop but ensures the loop body executes at least once before the condition is checked.
- **Loop Control Statements:**
  break exits the loop immediately, continue skips the current iteration, and goto transfers control to a labeled statement.

Structure of a for Loop in C

In C programming, the for loop has the following general structure:

```
for (initialization; condition; increment) {
   // Code to be executed on each iteration
}
```

Structure of a while Loop in C

In C programming, the while loop has the following general structure:

```
while (condition) {
   // Code to be executed on each iteration
}
```

Structure of a do-while Loop in C

The general structure of a do-while loop in C is:

```
do {
   // Code to be executed on each iteration
} while (condition);
```

Structure of Nested Loops

In most programming languages, including C, nested loops are structured as follows:

```
for (int i = 0; i < n; i++) {          // Outer loop
   for (int j = 0; j < m; j++) {       // Inner loop
      // Code to be executed
```

```
    }
}
```

Application of learning 2.4.

you are organizing a 3-day conference in ATL hotel Kigali with multiple sessions each day. Each session has multiple activities, such as presentations, workshops, and discussions. You need to create a schedule that covers all sessions and their activities over the 3 days. To do this, you'll need to use nested loops to iterate through each day, each session of the day, and each activity in the session. Write a c program make these activities.

Indicative content 2.5: Applications of functions

Duration: 5 hrs

Theoretical Activity 2.5.1: Description of types of C functions

Tasks:

1: Read carefully the given scenario and respond to the following questions:

I. What is a function in programming, and why do we use them?
II. Can you give an example of a situation in everyday life that's similar to how a function works in C?
III. What are some common functions you've encountered or used in C programming?
IV. How do you call a function in C, and what does it mean to "call" a function?
V. What is the purpose of a return statement in a function, and how is it used?

2: Discussion and write the findings on paper

3: Present their findings to the whole class.

4: Ask questions or concerns.

5: orients them to read the key reading **2.5.1**

Key readings 2.5.1. Description of types of C functions

Functions in C

Functions in C are self-contained blocks of code designed to perform specific tasks. They help in breaking down complex problems into smaller, manageable pieces, promoting code reuse, readability, and maintainability. Functions can take inputs, perform actions, and return results.

Types of Functions

- **Library Functions**: Predefined functions provided by C standard libraries (e.g., printf(), scanf(), strcpy()).
- **User-Defined Functions**: Custom functions created by the programmer to perform specific tasks.

Syntax

The basic structure of a function in C consists of:

1. **Function Declaration (Prototype)**: Tells the compiler about the function's name, return type, and parameters.
2. **Function Definition**: Contains the actual body of the function.

3. **Function Call**: Executes the function.

- **Function Declaration**
  A function declaration informs the compiler about the function name, return type, and parameters without providing the function body.
  return_type function_name(parameter_list);
  Example:
  int add(int a, int b);
- **Function Definition**
  A function definition includes the actual implementation of the function.
  return_type function_name(parameter_list) {
  // Function body
  }
  Example:
  int add(int a, int b) {
  return a + b;
  }
- **Function Call**
  A function call executes the function with specified arguments.
  function_name(argument_list);
  Example:
  int result = add(5, 3); // Calls the add function with arguments 5 and 3
  Example: Full Program with Functions

```
#include <stdio.h>

// Function declaration
int add(int a, int b);
void print_result(int result);

int main() {
   int num1 = 5, num2 = 3;
   int sum;

   // Function call
   sum = add(num1, num2);

   // Function call
   print_result(sum);

   return 0;
}
```

```
// Function definition
int add(int a, int b) {
   return a + b;
}

// Function definition
void print_result(int result) {
   printf("The result is: %d\n", result);
}
```

Output

csharp

Copy code

```
The result is: 8
```

Explanation

- ✓ **Function Declaration**: int add(int a, int b); and void print_result(int result); declare the functions.
- ✓ **Function Call**: sum = add(num1, num2); calls the add function, and print_result(sum); calls the print_result function.
- ✓ **Function Definition**: The add function computes the sum of two integers, and the print_result function prints the result.

Return Type

- **Void**: Indicates the function does not return a value.
- **Non-Void**: Indicates the function returns a value of a specified type (e.g., int, float, char).

Parameter Passing

Parameters are passed to functions by value in C, meaning the function gets a copy of the argument values.

Example:

```
void increment(int a) {
   a++;
}
```

Example: Swapping Two Numbers Using a Function

```
#include <stdio.h>

void swap(int *x, int *y);

int main() {
   int a = 5, b = 10;
```

```
    printf("Before swap: a = %d, b = %d\n", a, b);
    swap(&a, &b); // Pass addresses of a and b
    printf("After swap: a = %d, b = %d\n", a, b);

    return 0;
}

void swap(int *x, int *y) {
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

Output

less

Copy code

Before swap: a = 5, b = 10

After swap: a = 10, b = 5

Explanation

- **Pointer Parameters**: swap(int *x, int *y) uses pointers to swap the values of a and b.
- **Function Call**: swap(&a, &b) passes the addresses of a and b to the swap function.

Advantages of Using Functions

- ✓ **Modularity**: Breaks down complex problems into simpler, smaller functions.
- ✓ **Reusability**: Functions can be reused across different programs or within the same program.
- ✓ **Readability**: Makes the code easier to read and understand.
- ✓ **Maintainability**: Easier to maintain and debug.

Key Points

- Functions can have zero or more parameters.
- Functions can return a value or be void.

  Function declarations are necessary if the function is defined after the main function.
- Functions help in organizing code efficiently and promoting good programming practices.

Practical Activity 2.5.2: Applying C functions to solve a problem

Task:

1: Read carefully the given scenario and respond to the following questions:

You are developing a simple student grade management system for a school. The system needs to perform several tasks:

- **Add Student Grades:** Record grades for multiple subjects for each student.
- **Calculate Average Grades:** Compute the average grade for each student.
- **Determine Grade Category:** Categorize students based on their average grade (e.g., Excellent, Good, Average, Needs Improvement).

2: By using Reading **2.5.2**, Discussion and write the findings on paper

3: Present their findings to the whole class.

4: Ask questions or concerns.

Key readings 2.5.2 Applying C functions to solve a problem

C functions are not just fundamental to C programming but also pivotal in various practical applications. They enable modular programming, which simplifies code management and enhances functionality. Here are key applications and examples where C functions play a critical role:

1. Code Organization and Reusability

- **Modular Design**: Functions help in breaking down large programs into smaller, manageable modules, each responsible for a specific task.
- **Code Reuse**: Functions can be reused across different parts of a program or in different programs, reducing redundancy and making code maintenance easier.

Example:

```
#include <stdio.h>
// Function to compute the average
float compute_average(int a, int b, int c) {
   return (a + b + c) / 3.0;
}
int main() {
   int x = 10, y = 20, z = 30;
   float average = compute_average(x, y, z);
   printf("Average: %.2f\n", average);
   return 0;
}
```

2. Encapsulation of Functionality

- **Encapsulation**: Functions encapsulate specific tasks or operations, making the code modular and easier to understand. For example, separate functions can handle user input, processing, and output.

Example:

```
#include <stdio.h>
// Function to get user input
void get_input(int *num1, int *num2) {
   printf("Enter two integers: ");
   scanf("%d %d", num1, num2);
}
// Function to print the result
void print_result(int sum) {
   printf("The sum is: %d\n", sum);
}
int main() {
   int a, b, result;
   get_input(&a, &b);
   result = a + b;
   print_result(result);
   return 0;
}
```

3. Handling Complex Algorithms

- **Algorithm Implementation**: Functions are essential for implementing complex algorithms, such as sorting, searching, and mathematical computations, by breaking them into simpler, reusable parts.

Example:

```
#include <stdio.h>

// Function to find the maximum of two numbers
int max(int a, int b) {
   return (a > b) ? a : b;
}
// Function to perform binary search
int binary_search(int arr[], int size, int key) {
   int left = 0, right = size - 1;
   while (left <= right) {
      int mid = left + (right - left) / 2;
      if (arr[mid] == key) return mid;
      else if (arr[mid] < key) left = mid + 1;
```

```
        else right = mid - 1;
    }
    return -1; // Key not found
}

int main() {
    int arr[] = {1, 3, 5, 7, 9};
    int size = sizeof(arr) / sizeof(arr[0]);
    int key = 7;
    int index = binary_search(arr, size, key);
    printf("Element %d is at index %d\n", key, index);
    return 0;
}
```

4. Improving Program Readability

- **Readability**: Functions with descriptive names and clear purposes make the program easier to read and understand. They help in maintaining a clean and organized code structure.

Example:

```
#include <stdio.h>
// Function to calculate the area of a rectangle
float calculate_area(float length, float width) {
    return length * width;
}

// Function to display the area
void display_area(float area) {
    printf("The area of the rectangle is: %.2f\n", area);
}
int main() {
    float length = 5.0, width = 3.0;
    float area = calculate_area(length, width);
    display_area(area);
    return 0;
}
```

5. Error Handling and Validation

- **Validation**: Functions can be used to validate inputs and handle errors, ensuring the program behaves correctly under various conditions.

Example:

```
#include <stdio.h>
// Function to validate age
```

```
int validate_age(int age) {
    if (age < 0 || age > 120) {
        return 0; // Invalid age
    }
    return 1; // Valid age
}
int main() {
    int age;
    printf("Enter your age: ");
    scanf("%d", &age);

    if (validate_age(age)) {
        printf("Age is valid.\n");
    } else {
        printf("Invalid age entered.\n");
    }
    return 0;
}
```

6. Recursive Functions

- **Recursion**: Functions that call themselves can solve problems that can be broken down into smaller, similar sub-problems, such as computing factorials or generating Fibonacci sequences.

Example:

```
#include <stdio.h>

// Recursive function to compute factorial
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
int main() {
    int number = 5;
    printf("Factorial of %d is %d\n", number, factorial(number));
    return 0;
}
```

Points to Remember

Function in c

A function in C is a block of code designed to perform a specific task, helping organize and manage code effectively.

- **Library Functions:**
  Predefined functions provided by C libraries, such as scanf(), printf(), and sqrt().
- **User-Defined Functions:**
  Functions created by the programmer, involving a declaration (prototype), call, and definition, promoting code modularity and reuse.
- **Passing Parameters:**
  Parameters are declared in the function definition, while arguments are the values passed during function calls.
- **Calling Methods:**
  Call by Value passes a copy of the argument's value, while Call by Reference passes the argument's address, affecting the original argument.

To apply functions effectively in C:

- **Modular Design:** Break programs into smaller, manageable functions.
- **Parameter Passing:**
  - ✓ Use **pass-by-value** for simple types.
  - ✓ Use **pass-by-reference** (via pointers) for complex types or to modify data.
- **Return Values:** Use return to send a result back. Use void if no value is returned.
- **Error Handling:** Include checks (e.g., for division by zero) within functions.
- **Documentation:** Comment on functions to explain their purpose and parameters, enhancing readability and maintainability

Application of learning 2.5.

LDT Company located in Karongi district, need to make a calculator. Write a C program to create a calculator that illustrates the following tasks:

- Defining functions
- Declaring functions
- Calling functions

Indicative content 2.6: Application of pointers

Duration: 5 hrs

Practical Activity 2.6.1: Accessing the address of variables

Task:

1: Read carefully the given situation and perform tasks described below:

Envision you are creating a basic library management system. There are two variables: one for holding the total count of books in the library (int totalBooks), and another for storing the average book rating (float avgRating). To better understand how the system handles data in memory, you need to determine the memory locations of these variables.

Your task is to: Utilize pointers to store the addresses of these variables. Display the memory addresses by using the pointers.

2: By using the key **reading 2.6.1.,** Discuss how to access a variable pointer.

3: Write down the findings

4: Present their finding to the class.

5: Ask clarification where necessary.

Key readings 2.6.1 Accessing the address of variables

Pointers are a fundamental concept in C and C++ programming that allow you to store the address of a variable. This handout will guide you through the steps to declare pointers, assign addresses to them, and access the address and value of variables through pointers.

Key Concepts

- **Pointer**: A variable that stores the memory address of another variable.
- **Address-of Operator (&)**: Used to get the address of a variable.

- **Dereferencing Operator (*)**: Used to access the value at the address stored in a pointer.

  Steps to Access the Address of Variables Using Pointers

  Step 1: Declare Variables

  Create two variables of different data types. For example:

  int totalBooks = 100;

  float avgRating = 4.5;

  step 2: Declare Pointers

  Declare pointers corresponding to the data types of the variables:

  int *intPtr;

  float *floatPtr;

  Step 3: Assign Addresses to Pointers

  Assign the address of each variable to its respective pointer using the address-of operator (&):

  intPtr = &totalBooks;

  floatPtr = &avgRating;

  Step 4: Print Addresses Using Pointers

  Use the pointers to print the addresses of the variables:

  printf("Address of totalBooks: %p\n", (void*)intPtr);

  printf("Address of avgRating: %p\n", (void*)floatPtr);

  Step 5: Access and Print Values Using Pointers

  Use the dereferencing operator (*) to access and print the values stored at the addresses:

  printf("Value of totalBooks using pointer: %d\n", *intPtr);

  printf("Value of avgRating using pointer: %.2f\n", *floatPtr);

  Example Code

  #include <stdio.h>

  int main() {

```
    int totalBooks = 100;
    float avgRating = 4.5;
    // Declare pointers
    int *intPtr;
    float *floatPtr;
    // Assign addresses
    intPtr = &totalBooks;
    floatPtr = &avgRating;
    // Print addresses
    printf("Address of totalBooks: %p\n", (void*)intPtr);
    printf("Address of avgRating: %p\n", (void*)floatPtr);
    // Access and print values using pointers
    printf("Value of totalBooks using pointer: %d\n", *intPtr);
    printf("Value of avgRating using pointer: %.2f\n", *floatPtr);
    return 0;
}
```

Practical Activity 2.6.2: Declaring and initializing of pointers

Task:

1: Read carefully the given scenario and perform tasks described below:

In a student management system, you need to keep track of various data points, such as a student's ID and their GPA. You decide to use pointers to efficiently manage and manipulate this data in memory. Create variables for the student's ID and GPA, then declare and initialize matching pointers to their addresses. Use the pointers to print the variables' values and memory addresses, demonstrating data access and manipulation.

2: By using the key **reading 2.6.2,** Discuss on pointers declaration and initialization.

3: Write down the findings.

4: Present their finding to the class.

5: Ask clarification where necessary.

6: Read key **reading 2.6.2 Trainee Manual**

Key readings 2.6.2 Declaring and initializing of pointers

1. Definition

- **Pointer Declaration**: Specifies the data type of the variable the pointer will point to.
- **Pointer Initialization**: Assigns the address of a variable to a pointer.
- **Dereferencing**: Accessing the value stored at the address the pointer points to.

Steps to Declare and Initialize Pointers

Step 1: Declare Variables

Create variables of different data types:

int studentID = 12345;

float studentGPA = 3.75;

Step 2: Declare Pointers

Declare pointers for each variable, ensuring the pointer type matches the data type of the variable:

int *idPtr;

float *gpaPtr;

Step 3: Initialize Pointers

Assign the address of each variable to the corresponding pointer using the address-of operator (&):

idPtr = &studentID;

gpaPtr = &studentGPA;

Practical Activity 2.6.3: Accessing a variable through its pointer

Task:

1: Read carefully the given scenario and perform tasks described below:

In a grading system, you need to update a student's exam score stored in a variable. To practice using pointers, start by creating an int variable called examScore to hold the student's score. Then, declare a pointer int *scorePtr and assign it the address of examScore using the address-of operator (&). Finally, use the pointer to modify the value of examScore and print the updated value through the pointer to confirm the change, demonstrating how pointers can be used for direct data manipulation.

2: By using the key **reading 2.6.3** Discuss on Accessing a variable through its pointer.
3: Write the findings.
4: Present their finding to the class.
5: Ask clarification where necessary.

Key readings 2.6.3 Accessing a variable through its pointer

0. Introduction

In C and C++, a pointer is a variable that stores the memory address of another variable. Pointers provide a powerful mechanism for indirect access to variables and allow for more efficient memory management, especially when working with arrays, dynamic memory allocation, and complex data structures.

In this section we will focus on how to access a variable through its pointer. This concept is crucial to understanding how pointers work and how they can be used to modify and retrieve values stored in memory.

1. Pointers: A Refresher

Before diving into accessing variables through pointers, let's quickly review the basics of pointers.

1.1 What is a Pointer?

A pointer is a variable that stores the memory address of another variable. For example:

```
int x = 10;     // A normal integer variable

int *p = &x;    // p is a pointer to an integer, storing the address of x
```

Here:

- x is a regular integer.
- p is a pointer to an integer (int *), and it holds the address of x (obtained using the address-of operator &).

1.2 Declaring a Pointer

To declare a pointer, we specify the type of data it will point to followed by an asterisk (*), like so:

```
int *p;  // Pointer to an integer

char *c; // Pointer to a character

float *f; // Pointer to a float
```

1.3 The Address-of Operator (&)

The address-of operator (&) is used to get the memory address of a variable. For example:

```
int x = 10;

int *p = &x;  // &x gives the address of x
```

2. Dereferencing a Pointer

To access the value stored at the memory address held by a pointer, we use the dereference operator (*). Dereferencing a pointer allows us to access or modify the value at the address it points to.

2.1 Dereferencing Syntax

The general syntax for dereferencing a pointer is:

```
*pointer_name
```

This gives us the value stored at the address the pointer is pointing to.

2.2 Example: Dereferencing a Pointer

```
#include <stdio.h>

int main() {

   int x = 10;    // Declare an integer variable
```

```
    int *p = &x;   // Declare a pointer p that stores the address of x
    printf("Value of x: %d\n", x);      // Prints the value of x
    printf("Value via pointer: %d\n", *p);  // Dereference p to get the value of x
    return 0;
}
```

Output:

```
Value of x: 10
Value via pointer: 10
```

In the example above:

- x is a regular variable holding the value 10.
- p is a pointer storing the address of x.
- By dereferencing the pointer (*p), we get the value of x (which is 10).

2.3 Modifying a Variable Through Its Pointer

Since pointers give us direct access to a variable's memory, we can also modify the value of a variable through its pointer:

```
#include <stdio.h>
int main() {
    int x = 10;
    int *p = &x;

    printf("Before modification: %d\n", x); // Prints the original value of x

    *p = 20;  // Modify the value of x through the pointer

    printf("After modification: %d\n", x);  // Prints the updated value of x

    return 0;
}
```

Output:

mathematica

Copy code

Before modification: 10

After modification: 20

Here, *p = 20; modifies the value of x through the pointer p.

---

3. Pointer Arithmetic

Pointers allow you to perform arithmetic operations on the address they store. This is especially useful when working with arrays or dynamically allocated memory.

3.1 Incrementing a Pointer

You can increment or decrement a pointer to move it to the next or previous memory location of the type it points to. For example:

int arr[] = {10, 20, 30, 40};

int *p = arr; // p points to the first element of arr

printf("First element: %d\n", *p); // Dereference p to get the first element (10)

p++; // Move the pointer to the next element

printf("Second element: %d\n", *p); // Dereference p to get the second element (20)

In this case:

- p++ increments the pointer to the next element in the array arr.

3.2 Using Pointer Arithmetic to Traverse Arrays

Pointer arithmetic can be used to traverse arrays efficiently:

#include <stdio.h>

int main() {

  int arr[] = {10, 20, 30, 40};

  int *p = arr;

```
    for (int i = 0; i < 4; i++) {
        printf("Element %d: %d\n", i + 1, *(p + i));  // Dereference pointer with offset
    }

    return 0;
}
```

Output:

mathematica

Copy code

Element 1: 10

Element 2: 20

Element 3: 30

Element 4: 40

4. Pointers to Pointers

A pointer can also point to another pointer, creating a "pointer to a pointer." This is useful in scenarios involving multidimensional arrays or working with dynamic memory allocation.

4.1 Declaring a Pointer to a Pointer

To declare a pointer to a pointer, you use two asterisks (**):

```
int x = 10;
int *p = &x;
int **pp = &p;  // Pointer to pointer
```

4.2 Accessing the Value Through a Pointer to a Pointer

To access the value of x through pp, you first dereference pp to get p, then dereference p to get the value of x:

```
#include <stdio.h>

int main() {
```

```
    int x = 10;
    int *p = &x;
    int **pp = &p;

    printf("Value of x: %d\n", **pp);  // Dereference pp to get the value of x

    return 0;
}
```

Output:

Value of x: 10

5. Common Pointer Operations

Here are some common operations when working with pointers:

- **Getting the address of a variable:**

  ```
  int x = 5;
  int *p = &x;  // &x gives the memory address of x
  ```

- **Dereferencing a pointer to access or modify the value:**

  ```
  *p = 10;  // Modify the value stored at the address p points to
  ```

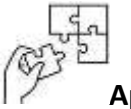
- **Pointer arithmetic (e.g., p++ to move to the next element in an array).**
- **Null pointers:** A null pointer is a pointer that does not point to any valid memory location. It is initialized to NULL:

  ```
  int *p = NULL;
  ```

Points to Remember

Data type in c programming

- Pointers in C and C++ store the memory address of a variable, allowing indirect access to its value.
- The address-of operator (&) is used to get a variable's address, and the dereferencing operator (*) is used to access its value through a pointer.
- Steps include declaring variables, declaring pointers, assigning addresses, and using pointers to print addresses and access values.
- Pointers store memory addresses: A pointer is a variable that holds the memory address of another variable, allowing indirect access to its value.
- Declaring pointers: Pointers are declared using an asterisk (*) followed by the variable type (e.g., int *p for a pointer to an integer).
- Address-of operator (&): The & operator is used to obtain the memory address of a variable, which can then be stored in a pointer.
- Dereferencing a pointer: The * operator is used to access or modify the value stored at the memory address held by a pointer.
- Modifying variables through pointers: Pointers allow direct modification of the value stored at the memory address, enabling changes to variables indirectly.
- Pointer arithmetic: Pointers can be incremented or decremented to traverse memory locations, which is especially useful for working with arrays or dynamic memory.



Application of learning 2.6.

In this scenario, you are developing a program to manage student data, focusing on their scores in two subjects using pointers for efficient data manipulation. Begin by declaring two integer variables, subject1 and subject2, and initializing them with scores such as 85 and 90. Next, declare two integer pointers, ptr1 and ptr2, to store the addresses of these variables. Assign the addresses of subject1 and subject2 to ptr1 and ptr2 using the address-of operator (&). Use these pointers to print the memory addresses and values of subject1 and subject2. Modify the scores by using ptr1 and ptr2 to increase subject1 by 5 and subject2 by 10, then print the updated values. Lastly, declare an integer array studentScores to hold scores for four subjects, and use a pointer to traverse the array, incrementing the pointer to access and print each score.

Indicative content 2.7: Applications of arrays

Duration: 10 hrs

Theoretical Activity 2.7.1: Describe the Types of arrays

Tasks:

1: Read carefully the given scenario and respond to the following questions:

i. What is an array in programming, and why is it used?
ii. How are elements stored in an array, and what is the significance of the index?
iii. What are the benefits of using arrays in your code?
iv. What are some common use cases where arrays are indispensable?

2: Discuss and write the findings on paper

3: Present their findings to the whole class.

4: Ask questions or concerns.

5: Read the key reading **2.7.1**

Key readings 2.7.1.: Describe the Types of arrays

Arrays in c Programming:

In C programming, arrays are used to store multiple values of the same type in a contiguous block of memory. They are a fundamental data structure that allows for efficient access and manipulation of elements. There are several types of arrays, each serving different purposes:

1. Single-Dimensional Arrays

Definition: The simplest form of an array, where elements are stored in a linear sequence.

Declaration and Initialization:

```
// Declaration
int numbers[5];

// Initialization
int numbers[5] = {1, 2, 3, 4, 5};
```

Accessing Elements:

```
int first = numbers[0]; // Accesses the first element
int second = numbers[1]; // Accesses the second element
```

Use Case: Ideal for storing lists of values like student grades, temperatures, or any other one-dimensional data.

2. Multi-Dimensional Arrays

Definition: Arrays with more than one dimension, often used to represent matrices or tables.

2.1. Two-Dimensional Arrays

Declaration and Initialization:

```
// Declaration
int matrix[3][4];

// Initialization
int matrix[3][4] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8},
   {9, 10, 11, 12}
};
```

Accessing Elements:

```
int value = matrix[1][2]; // Accesses the element at row 1, column 2
```

Use Case: Useful for representing grid-like data structures such as chessboards, game maps, or spreadsheet data.

2.2. Three-Dimensional Arrays

Declaration and Initialization:

```
// Declaration
int cube[2][3][4];

// Initialization
int cube[2][3][4] = {
   {
      {1, 2, 3, 4},
      {5, 6, 7, 8},
      {9, 10, 11, 12}
   },
   {
      {13, 14, 15, 16},
      {17, 18, 19, 20},
      {21, 22, 23, 24}
   }
};
```

Accessing Elements:

int value = cube[1][2][3]; // Accesses the element in the second matrix, third row, fourth column

Use Case: Suitable for applications involving three-dimensional data, such as 3D graphics or volumetric data.

3. Jagged Arrays (Arrays of Arrays)

Definition: Arrays where each element is a different-sized array. They are also known as "array of arrays."

Declaration and Initialization:

```
// Declaration
int *jagged[3];

// Initialization
int row1[] = {1, 2, 3};
int row2[] = {4, 5};
int row3[] = {6, 7, 8, 9};
jagged[0] = row1;
jagged[1] = row2;
jagged[2] = row3;
```

Accessing Elements:

int value = jagged[1][1]; // Accesses the second element of the second row

Use Case: Useful for storing data where each row or sub-array can be of different sizes, such as varying-length records.

4. Dynamic Arrays

Definition: Arrays whose size can be determined at runtime. They are created using dynamic memory allocation functions like malloc(), calloc(), and realloc.

Declaration and Initialization:

```
#include <stdlib.h>

// Declaration
int *dynamicArray;

// Allocation
dynamicArray = (int *)malloc(5 * sizeof(int));

// Initialization
for (int i = 0; i < 5; i++) {
   dynamicArray[i] = i + 1;
}

// Free memory
```

free(dynamicArray);

Accessing Elements:

int value = dynamicArray[2]; // Accesses the third element

Use Case: Ideal for situations where the size of the array is not known at compile time and can change during program execution.

Summary

- **Single-Dimensional Arrays**: Simple linear data structure, ideal for lists and sequences.
- **Two-Dimensional Arrays**: Useful for grid-like data structures such as matrices.
- **Three-Dimensional Arrays**: Suitable for representing 3D data, such as volumes or 3D matrices.
- **Jagged Arrays**: Arrays of varying sizes, useful for non-uniform data.
- **Dynamic Arrays**: Arrays whose size can be changed at runtime, useful for flexible data storage.

Practical Activity 2.7.2: Declaring Arrays

Task:

1: Read carefully the given situation perform tasks described below:

You are a teacher managing a class of students. You need to store the grades of your students in a program so you can easily calculate averages, find the highest and lowest grades, and perform other analyses.

- Declare all the array in your program using a suitable syntax

2: By using the Reading **2.7.2 ,**Discuss by writing the syntax declaration of arrays.

3: write the code in a given syntax of arrays

4: present their finding to the class.

5: Ask clarification where necessary.

6: Reading **2.7.2**

Key readings 2.7.2 Declaring Arrays

In C programming, an array is a collection of elements of the same data type stored in contiguous memory locations. The declaration of arrays involves specifying the type of elements and the size of the array. Here's a detailed overview of array declaration:

1. Basic Syntax

The syntax for declaring an array is as follows:

type array_name[size];

- **type**: The data type of the elements (e.g., int, float, char).
- **array_name**: The name of the array.
- **size**: The number of elements in the array.

Example:

int numbers[5]; // Declares an array of 5 integers

2. Initialization at Declaration

Arrays can be initialized at the time of declaration. The syntax for initialization is:

type array_name[size] = {value1, value2, ..., valueN};

- **value1, value2, ..., valueN**: Initial values for the array elements.

Example:

int numbers[5] = {1, 2, 3, 4, 5}; // Declares and initializes an array of 5 integers

If the number of initializers is less than the size of the array, the remaining elements are initialized to zero:

int numbers[5] = {1, 2}; // {1, 2, 0, 0, 0}

If the number of initializers is greater than the size of the array, the excess initializers are ignored:

int numbers[3] = {1, 2, 3, 4}; // {1, 2, 3} (4 is ignored)

3. Omitting Size

The size of the array can be omitted if it is initialized at the same time. The size is automatically determined based on the number of initializers:

int numbers[] = {1, 2, 3, 4, 5}; // Size is 5

4. Multi-Dimensional Arrays

2D Arrays: The syntax for a 2D array is:

type array_name[rows][columns];

- **rows**: Number of rows.
- **columns**: Number of columns.

Declaration and Initialization:

```
int matrix[3][4] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8},
```

```
    {9, 10, 11, 12}
};
```

3D Arrays: The syntax for a 3D array is:

```
type array_name[depth][rows][columns];
```

Declaration and Initialization:

```
int cube[2][3][4] = {
    {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    },
    {
        {13, 14, 15, 16},
        {17, 18, 19, 20},
        {21, 22, 23, 24}
    }
};
```

5. Accessing Array Elements

Array elements are accessed using indices. The index starts from 0 for the first element and goes up to size-1 for the last element.

Example:

```
int numbers[5] = {1, 2, 3, 4, 5};
int first = numbers[0];   // Accesses the first element (1)
int second = numbers[1];  // Accesses the second element (2)
```

For multi-dimensional arrays:

```
int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
int value = matrix[1][2];  // Accesses the element at row 1, column 2 (7)
```

6. Default Values

- **Integer Arrays**: Uninitialized integer arrays contain garbage values unless explicitly initialized.
- **Character Arrays**: Uninitialized character arrays may contain random characters.
- **Floating-Point Arrays**: Uninitialized floating-point arrays may contain garbage values.

7. Size Determination

- **Size of a Single-Dimensional Array**:

```
int size = sizeof(numbers) / sizeof(numbers[0]);
```

This calculates the number of elements in the array numbers.

- **Size of Multi-Dimensional Arrays**:
  int rows = sizeof(matrix) / sizeof(matrix[0]);
  int columns = sizeof(matrix[0]) / sizeof(matrix[0][0]);

Practical Activity 2.7.3: Initializing Arrays

Task:

1: Read carefully the given situation and perform task described below:
You are a teacher managing a class of students. You need to store the grades of your students in a program so you can easily calculate averages, find the highest and lowest grades, and perform other analyses.
2: By using the key reading **2.7.3,** Discuss by writing the code for array initialization.
3: write the code in a given syntax of arrays
4: Present their finding to the class.
5: Ask clarification where necessary.

Key readings 2.7.3: Initializing Arrays

Array initialization in C

In C programming, arrays can be initialized in several ways, depending on the type of array and the requirements of the program. Here's a detailed guide on how to initialize arrays:

1. Single-Dimensional Arrays

1.1. Direct Initialization

You can initialize an array at the time of declaration by specifying a list of values in curly braces {}. The size of the array can be explicitly specified or omitted.

- **Explicit Size:**
  int numbers[5] = {1, 2, 3, 4, 5}; // Size is explicitly set to 5
- **Implicit Size:**
  int numbers[] = {1, 2, 3, 4, 5}; // Size is automatically set to 5 based on the number of initializers

If fewer initializers are provided than the size of the array, the remaining elements are initialized to zero:
int numbers[5] = {1, 2}; // {1, 2, 0, 0, 0}
If more initializers are provided than the size, the excess values are ignored:

```
int numbers[3] = {1, 2, 3, 4};  // {1, 2, 3} (4 is ignored)
```

1.2. Initialization with Specific Values

You can specify values for individual elements after declaring the array:

```
int numbers[5];
numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 3;
numbers[3] = 4;
numbers[4] = 5;
```

2. Multi-Dimensional Arrays

2.1. Two-Dimensional Arrays

You can initialize a 2D array by providing initializers for each row:

```
int matrix[3][4] = {
   {1, 2, 3, 4},
   {5, 6, 7, 8},
   {9, 10, 11, 12}
};
```

If fewer initializers are provided for any row, the remaining elements in that row are initialized to zero:

```
int matrix[3][4] = {
   {1, 2},
   {3, 4},
   {5, 6}
};  // {{1, 2, 0, 0}, {3, 4, 0, 0}, {5, 6, 0, 0}}
```

2.2. Three-Dimensional Arrays

Similarly, a 3D array can be initialized by providing initializers for each 2D "layer":

```
int cube[2][3][4] = {
   {
      {1, 2, 3, 4},
      {5, 6, 7, 8},
      {9, 10, 11, 12}
   },
   {
      {13, 14, 15, 16},
      {17, 18, 19, 20},
      {21, 22, 23, 24}
   }
};
```

If fewer initializers are provided, the remaining elements are zero-initialized:

c
Copy code

```
int cube[2][2][2] = {
   {
      {1, 2},
      {3, 4}
   }
};  // {{{1, 2}, {3, 4}}, {{0, 0}, {0, 0}}}
```

3. Array Initialization with Loops

You can also use loops to initialize arrays dynamically. This is useful when the array size is not known at compile time or when you want to initialize elements with a pattern.

Example:

```
#include <stdio.h>
int main() {
   int numbers[5];
   for (int i = 0; i < 5; i++) {
      numbers[i] = i * 2;  // Initialize each element with its index multiplied by 2
   }

   // Print the array
   for (int i = 0; i < 5; i++) {
      printf("%d ", numbers[i]);
   }

   return 0;
}
```

4. Partial Initialization

For arrays with more elements than initializers, the remaining elements are automatically set to zero:

```
int numbers[5] = {1, 2};  // Initializes as {1, 2, 0, 0, 0}
```

5. String Initialization

Strings in C are arrays of characters ending with a null terminator '\0'. They can be initialized with double quotes:

```
char str[] = "Hello";  // Automatically includes the null terminator
```

Points to Remember

Arrays in c Programming:

In C programming, arrays are data structures that store multiple elements of the same type in contiguous memory. The types of arrays include:

- **One-Dimensional Array:** A single row of elements. Declared with one pair of square brackets.
  - ✓ **Example:** int numbers[5];
- **Two-Dimensional Array:** An array of arrays, forming a matrix. Declared with two pairs of square brackets for rows and columns.
  - ✓ **Example:** int matrix[3][4];
- **Multi-Dimensional Array:** Extends beyond two dimensions with multiple levels of brackets.
  - ✓ **Example:** int data[2][3][4];

Arrays declaration in c

In C programming, arrays are declared by specifying the data type of the elements and the array size. The methods of declaration vary based on the array type:

- **One-Dimensional Array:**

data_type array_name[array_size];

- **Two-Dimensional Array:**

data_type array_name[rows][columns];

Syntax of Array Initialization

The general syntax for initializing an array during its declaration is as follows:

data_type array_name[array_size] = {value1, value2, value3, ...};

- data_type: The type of elements in the array (e.g., int, float, char).
- array_name: The name you give to the array.
- array_size: The number of elements the array can hold. This can sometimes be omitted if the array is being initialized with values.
- {value1, value2, value3, ...}: A comma-separated list of values used to initialize the array elements.

Application of learning 2.7.

You are a software developer working on a sales analysis application for a retail store. You need to record the sales figures for each day of the week and then calculate the total and average sales. To achieve this, you will declare and initialize an array with the sales data for the week.

Declare and initialize an array to store the daily sales figures for a week (7 days) and then calculate and display the total and average sales.

Learning outcome 2 end assessment

Theoretical assessment

Choose the letter corresponding to the correct answer:

1. What is the correct syntax for declaring a variable in C?

A.var int x;

B. int x;

C. variable int x;

D. int x = var;

2. Which of the following is the correct way to include a header file in a C program?

A. #include "stdio.h"

B. include <stdio.h>

C. #include <stdio.h>

D. #include <stdio.h.h>

3. **What will be the output of the following C code snippet?**

```
#include <stdio.h>
int main() {
   int a = 10, b = 5;
   printf("%d\n", a + b);
   return 0;
}
```

A. 10

B. 15

C. 5

D. 105

4. Which of the following data types can be used to store a floating-point number in C?

A. int

B. char

C. float

D. bool

5. What is the output of the following code?

```
#include <stdio.h>

int main() {
   int x = 7;
   printf("%d\n", x % 3);
   return 0;
}
```

A. 7
B. 2
C. 3
D. 1

6. How do you declare an array of 10 integers in C?

A. int arr[10];
B. int arr(10);
C. array int arr[10];
D. int arr = [10];

7. Which function is used to read input from the user in C?

A. scanf()
B. printf()
C. input()
D. read()

8. What is the purpose of the return 0; statement in the main() function?

A. To exit the program with a status code of 0
B. To restart the program
C. To print 0 to the console
D. To create a new integer variable

9. What is the correct syntax for defining a function in C?

A. return_type function_name(parameters) { /* code */ }
B. function return_type function_name(parameters) { /* code */ }
C. function return_type { /* code */ } function_name(parameters)
D. function_name(parameters) return_type { /* code */ }

10. What is a function in C programming?

A) A reserved keyword
B) A block of code that performs a specific task
C) A type of data structure
D) A type of loop

11. Which of the following is a valid way to declare a function in C?

A) int function_name[];
B) function_name int();
C) int function_name();
D) function_name = int();

12. Which keyword is used to return a value from a function in C?

A) void
B) return
C) break
D) exit

13. Which of the following is a valid control statement in C?

A) loop

B) do-while

C) foreach

D) switch-case

14. What is the output of the following code?

```
int x = 5;
if (x > 3) {
   printf("Hello");
} else {
   printf("World");
}
```

A) World

B) Hello

C) Error

D) 5

15. Which control statement allows a program to execute a block of code repeatedly based on a condition?

A) if

B) for

C) switch

D) goto

16. In C, which of the following statements is used to terminate a loop?

A) continue

B) break

C) return

D) switch

17. What does the continue statement do in a loop?

A) Terminates the loop

B) Skips the current iteration and continues with the next iteration

C) Returns a value

D) Ends the program

18. Which of the following is the correct syntax for a switch statement in C?

A) switch (variable) {}

B) if (variable) {}

C) loop (variable) {}

D) case (variable) {}

19. What is the purpose of the default case in a switch statement?

A) To execute when none of the other cases match

B) To stop the execution of the program

C) To execute before any other case

D) To handle all cases

Subjective questions

1. Explain the difference between a **for** loop and a **while** loop in C. Provide examples of when to use each.
2. What is an array in C, and how is it declared and initialized? Provide an example.
3. How does a **switch** statement work in C, and how does it differ from an **if-else** statement? Illustrate with code examples.
4. Describe the purpose of the **break** and **continue** statements in loops. Provide examples of how each is used.
5. Write a C program to find the largest number in an array of integers. Explain the logic used in your solution.
6. What are nested loops, and when are they used in C programming? Provide an example of a situation where nested loops would be necessary.
7. Discuss the concept of an infinite loop. How can an infinite loop be created in C, and how can it be terminated?
8. Explain how multidimensional arrays are used in C. Provide an example of how a 2D array can be used to store and access data.
9. What are the limitations of using arrays in C? How do arrays differ from other data structures like linked lists?
10. What are pointers in C, and why are they important?
11. Explain pointer arithmetic in C. How does it work with different data types?
12. What is the difference between a pointer to a pointer and a pointer to an array?

Practical assessment

1. Write a C program to find the sum of all integers from 1 to n, where n is provided by the user.
2. Write a C program to check if a given number is a prime number.
3. Write a C program to reverse a string entered by the user.
4. Write a C program to calculate the factorial of a non-negative integer using recursion.
5. Write a C program to find the largest and smallest number in an array of integers.
6. Write a C program to sort an array of integers in ascending order using the bubble sort algorithm.
7. Write a C program that swaps two integers using pointers. You need to create a function swap(int *a, int *b) that takes two integer pointers as arguments and swaps the values of the two integers. Demonstrate the function in the main() function by inputting two numbers and printing the swapped values.

References

T. F. L. Y. Presents, “Practical C Programming , 3rd Edition By Steve Oualline 3rd Edition August 1997 ISBN : Table of Contents Getting Help in an Integrated Development Environment,” no. August, pp. 1–70, 1997.

S. G. Kochan, *Programming in C Warning and Disclaimer Bulk Sales*. 2004.

S. C. Dewhurst and K. Stark, *Programming in C++*, vol. 2, no. 4. 1991. doi: 10.1145/126983.126989.

M. Vine, *C Programming for the Absolute Beginner*. 2008.

https://ccsuniversity.ac.in/bridge-library/pdf/btech-cs/Functions%2025,26,27,28,-converted.pdf

https://courses.minia.edu.eg/Attach/16036flowchart-algorithm-manual.pdf

https://engineerstutor.com/2020/10/05/solved-assignment-problems-in-c-with-algorithm-and-flowchart/.

https://intranet.cb.amrita.edu/sites/default/files/1.6%20Function.pdf

https://www.albany.edu/faculty/dsaha/teach/2017Spring_CEN360/slides/lec08.pdf

https://www.digitalocean.com/community/tutorials/initialize-an-array-in-c

https://www.geeksforgeeks.org/c-arrays/

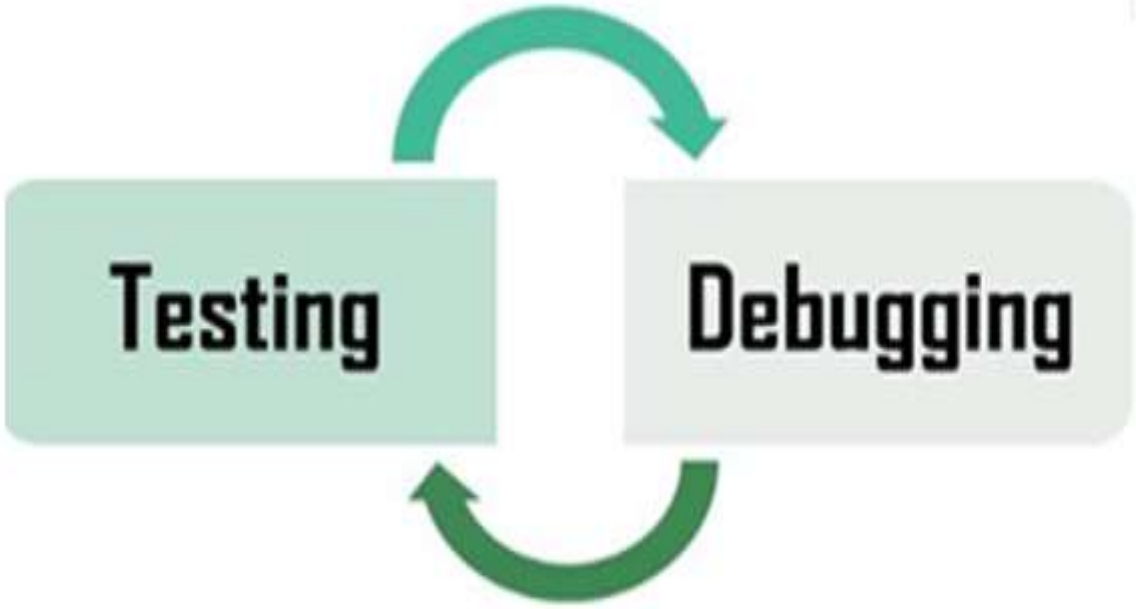
https://www.simplilearn.com/tutorials/c-tutorial/array-in-c#:~:text=Array%20in%20C%20can%20be,data%20types%20too%20like%20structures.

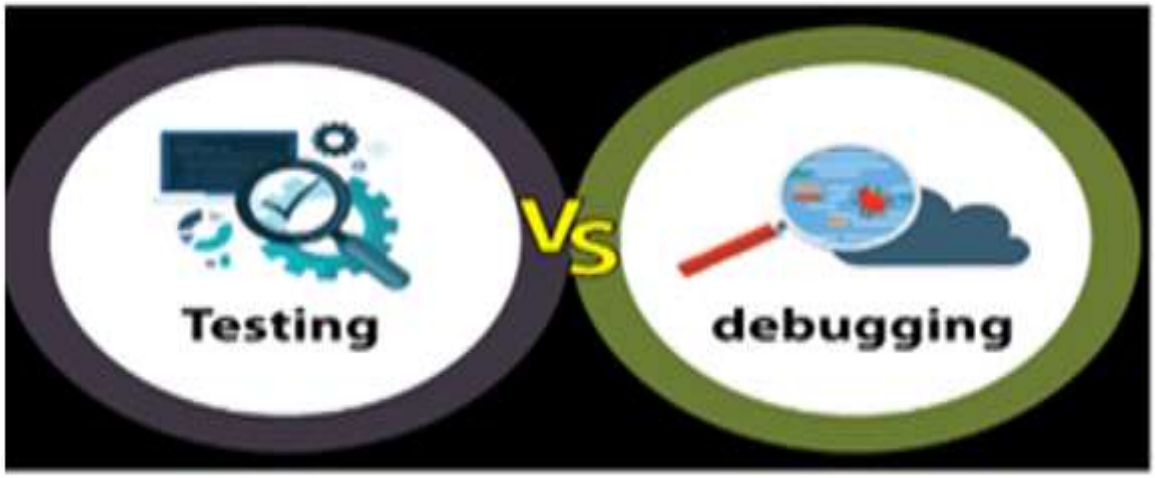
https://www.tutorialspoint.com/cprogramming/c_arrays.htm

https://www.tutorialspoint.com/cprogramming/pdf/c_functions.pdf

https://www.zenflowchart.com/guides/flowchart-in-c-programming.

https://www2.seas.gwu.edu/~bell/csci1121/lectures/functions.pdf


Testing
Debugging


Testing
Vs
debugging

Indicative contents
3.1. Identification of errors **3.2. Compilation of the C program** **3.3. Test of the C program**

Key Competencies for Learning Outcome 3: Perform program debugging

Knowledge	Skills	Attitudes
• Description of errors types. • Description of types of compiler • Description of debugging techniques and tools you can use when working with C programming • Description of the types of testing	• Selecting types of errors, including syntax errors, logical errors, and runtime errors in a c program • Applying debugging tools like GDB or integrated development environment (IDE) debuggers. • Debugging a c program • Selecting of different types of compilers in c program • Testing a c program	• Having an innovative • Having Creativity • Having Critical thinking • Having Teamwork • Being Problem Solver • Being Patient • Having Punctuality • Having Curiosity

Duration: 10hrs

Learning outcome 2 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly different program errors in computer programming

2. Compile C program in accordance with the program instructions.

3. Debug correctly the errors in a c program.

4. check errors correctly based on programming instructions.

5. Test correctly based on programming instructions.

Resources

Equipment	Tools	Materials
• Computers	• C programming IDE	• Internet

Indicative content 3.1: Identification of errors

Duration: 5 hrs

Theoretical Activity 3.1.1: Describe types of errors

Tasks:

1: You are requested to answer the following questions:

i. What is an error in c programming?
ii. How to read an error in c programming?
iii. What are the different types of errors?

2: Discuss to the questions asked in task 1 and write down the answers on paper

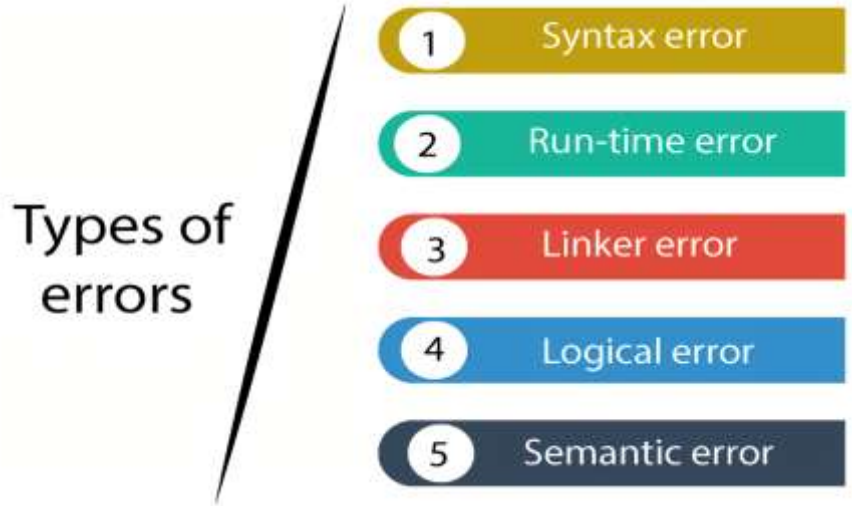
3: Present findings to the whole class.

4: For more clarification, if necessary

Key readings 3.1.1. Describe types of errors

Types of Errors in C Programming

- **An error in c programming:** An Error is an illegal operation performed by the user which results in abnormal working of the program.
- **Identify different types of errors in c programming**



Syntax errors: Errors that occur when you violate the rules of writing c

example:

```
#include <iostream>
using namespace std;

int main(void)
```

```
{
   // while() cannot contain "." as an argument.
   while(.)
   {
     printf("hello");
   }
   return 0; }
```

Run-time Errors : Errors which occur during program execution(run-time) after successful compilation are called run-time errors. One of the most common run-time errors is division by zero also known as Division error. These types of error are hard to find as the compiler doesn't point to the line at which the error occurs.
example:

```
#include <stdio>
#include <bits/stdc.h>
using namespace std;

void main()
{
   int n = 9, div = 0;

   // wrong logic
   // number is divided by 0,
   // so this program abnormally terminates
   div = n/0;

  printf("result =  %d",div);
}
```

linker error : errors generated when the executable of the program cannot be generated. This may be due to wrong function prototyping, incorrect header files. One of the most common linker errors is writing Main() instead of main().
example:

```
#include <bits/stdc.h>  //<bits/stdc.h> should be <stdio.h>
using namespace std;
void Main() // Here Main() should be main()
{
   int a = 10;
   printf( "a "); }
```

logical error : On compilation and execution of a program, desired output is not obtained when certain input values are given. These types of errors which provide incorrect output but appear to be error free are called logical errors.

example:

```
#include<stdio.h>
int main()
{int i = 0;
  // logical error : a semicolon after loop
  for(i = 0; i < 3; i++);
  {
        printf ("loop ");
        continue;
  }
  return 0; }
```

Semantic errors : This error occurs when the statements written in the program are not meaningful to the compiler.

example: a + b = c; //semantic error\

Practical Activity 3.1.2: Correcting errors

Task:

1: Read the following scenario and do the following task.

In your computer lab, suppose that you're working on a simple C program that reads a series of numbers from the user, calculates their sum, and then displays the result. However, there are several errors in the code, and your task is to identify practical techniques for correcting errors and correct them.

2: By using the Key readings **3.1.2**, Discuss to the scenario in task above

3: Perform the activity above in your computer

4: Identify the founded errors during the activity, if any

5: Correct the identified errors and compile.

6: Present your work to the whole class.

7: Ask For more clarification.

Key readings 3.1.2 Correcting errors

Correct error in c programming

Correcting errors in C programming involves a combination of techniques and strategies to identify and fix issues in your c code.

- **Techniques of correcting errors in c programming**

  Correcting errors in C programming involves a systematic and structured approach to identifying and fixing issues in your code.

- **Here are some techniques to help you correct errors in C programming:**

✓ **Understanding the Error Messages:**

Pay close attention to compiler and runtime error messages. They often provide valuable information about the nature and location of the error. Read the messages carefully to understand what went wrong.

✓ **Inspect the Code:**

Review your code thoroughly, line by line, to identify syntax errors, logical errors, or potential issues. Compare your code against the requirements and expectations.

✓ **Use a Debugger:**

Debugging tools like GDB (GNU Debugger) can help you step through your code, inspect variable values, and identify the exact location of errors. Set breakpoints in your code to stop execution at specific points.

✓ **Print Statements:**

Insert print statements (printf in C) strategically throughout your code to output variable values, program flow, and diagnostic messages. This can help you trace the program's execution and identify issues.

✓ **Isolate the Problem:**

If you encounter a runtime error, try to isolate the problem by narrowing down the section of code causing the issue. Comment out parts of the code or use a binary search approach to identify the problematic section.

✓ **Divide and Conquer:**

For logical errors, break your code into smaller components and test each part independently. This "divide and conquer" approach makes it easier to pinpoint the source of the error.

✓ **Check for Typographical Errors:**

Syntax errors often result from typographical mistakes, such as missing semicolons, mismatched parentheses, or typos in variable and function names. Carefully review your code for such errors.

✓ **Variable Initialization:**

Ensure that all variables are properly initialized before use, especially if they are being used for calculations. Uninitialized variables can lead to undefined behavior.

- ✓ **Data Type Mismatches:**
  Pay attention to data type mismatches in expressions and function calls. Make sure that the types of operands are compatible and that you're using the correct format specifiers in printf and scanf.
- ✓ **Memory Management:**
  For issues related to memory allocation and deallocation, use memory management functions like malloc, free, calloc, and realloc correctly. Be diligent in releasing allocated memory to prevent memory leaks.
- ✓ **Array Bounds Checking:**
  If you're working with arrays, ensure that you're not accessing elements outside the array bounds. Array index errors can lead to buffer overflows or segmentation faults.
- ✓ **Avoiding Infinite Loops:**
  Carefully inspect your loops to prevent infinite loops. Make sure your loop control variables are being updated correctly and that loop termination conditions are met.
- ✓ **Error Handling:**
  Implement error-handling mechanisms to gracefully handle exceptions and unexpected conditions, especially in functions that may encounter errors during execution.
- ✓ **Peer Review:**
  Collaborate with peers or colleagues to conduct code reviews. A fresh pair of eyes can often spot errors that you may have missed.
- ✓ **Documentation and Comments:**
  Maintain clear and concise code documentation and add comments to explain the purpose and logic of your code. This can aid in understanding and debugging your code.
- ✓ **Testing and Validation:**
  Thoroughly test your code with various test cases, including boundary cases and edge cases. Ensure that your code produces the expected results.
- ✓ **Learning from Mistakes:**
  Embrace errors as opportunities for learning. Take note of common mistakes you make and strive to avoid them in the future.
- ✓ **Use Static Code Analysis Tools:**
  Utilize static code analysis tools that can automatically identify potential issues in your code and offer suggestions for improvement.Correcting errors in C programming is an essential skill that requires patience, attention to detail, and a methodical approach.

Points to Remember

Types of Errors in C Programming

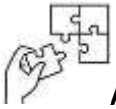
In C programming, an error refers to any deviation from the intended or expected behavior of a program. Errors can occur at various stages of development and execution and are categorized into three main types: syntax errors, logical errors, and runtime errors.

- **Syntax Errors:**
  - ✓ Violations of the rules of the C programming language. They occur when the code does not conform to the syntax of C.

- **Logical Errors:**
  - ✓ Flaws in the logic or design of a program, leading to undesired outcomes or
- **Runtime Errors:**
  - ✓ Occur during the execution of a program due to unexpected conditions.

Correcting Errors in C Programming

- Understanding Error Messages
- Inspect the Code
- Use a Debugger
- Print Statements
- Isolate the Problem
- Divide and Conquer



Application of learning 3.1.

you are an engineer working on an embedded system project for a weather monitoring station. The station collects temperature readings from multiple sensors installed in different locations (e.g., on different floors of a building or at various points in a large area). Your task is identify the errors in the segment of code

```
#include <stdio.h>
int main() {
   int arr[5] = {1, 2, 3, 4, 5};
   for (int i = 0; i <= 5; i++) {
      printf("%d ", arr[i]);
   }
   return 0;
}
```

Task:

1. Identify the error in the code and explain why it occurs.
2. How would you correct the error to prevent the program from accessing out-of-bounds memory?

Indicative content 3.2: Compilation of The C Program

Duration: 3 hrs

Theoretical Activity 3.2.1: Describe types of compilers

Tasks:

1: You are requested to answer the following questions:

I. What is a compiler?
II. What does the compiler do exactly?
III. identify the different types of compilers

2: Discuss to the questions asked in step 1and write down the answers on paper

3: Present findings to the whole class.

4: Ask questions for more clarification, if any.

5: Read the Key readings 3.2.1

Key readings 3.2.1. Describe types of compilers

Types of compilation

- **What is a Compiler?**

  A **compiler** is software that converts high-level language code (like C) into machine-level language. It checks for errors in the code and lists them if the input code doesn't adhere to language rules.

What Does the Compiler Do?

- **Source Code Analysis**: Reads and analyzes the source code structure and syntax (lexical analysis and parsing).
- **Syntax Checking**: Checks for syntax errors and reports them to the developer.
- **Semantic Analysis**: Ensures the code makes logical sense within the programming language context, checking data types, variable declarations, and function calls.
- **Intermediate Code Generation**: Generates an intermediate representation of the code, which is platform-independent and easier to optimize.
- **Optimization**: Applies techniques to improve the efficiency of the generated machine code.
- **Code Generation**: Translates the source code into machine or assembly code specific to the target computer architecture.
- **Linking**: If necessary, links multiple modules to create the final executable program.

- **Executable Output**: Produces an executable file containing machine code that the computer can execute.

Types of Compilers:

- **Borland Turbo C**: A popular and basic C compiler known for its small size and fast compilation speed, re-released as freeware by Embarcadero Technologies in 2006.
- **Tiny C Compiler (TCC)**: Designed for performance on slow computers with limited disk space, it is known for its small size and fast speed, approximately nine times faster than GCC.
- **Portable C Compiler (PCC)**: An early and influential compiler, prevalent in the mid-1970s, designed with machine-dependent source files for better portability and error checking.
- **GCC (GNU Compiler Collection)**: Supports multiple languages, initially released for C, and is known for extensive optimizations, such as dead code elimination and redundancy removal.
- **Clang**: Supports C, C++, Objective-C, and Objective-C++, known for its detailed error reports and memory efficiency, and uses LLVM for backend compilation.
- **MSVC (Microsoft Visual C++)**: A commercial compiler integrated with Visual Studio, supporting C and C++, known for Windows-specific features and tools.
- **ICC (Intel C++ Compiler)**: Optimized for Intel processors, supports C and C++, and offers high performance and parallelism.
- **MinGW (Minimalist GNU for Windows)**: A port of GCC for Windows, providing a minimal set of tools to compile and run C programs on Windows, known for its lightweight and fast performance

Practical Activity 3.2.2: Compilation techniques

Task:

1: Read the following scenario and do the following task.

Compile and run a simple C program that prints "Hello, World!" to the console.

2: By using the Key readings **3.2.2,** write the source code

3: Open terminal/command prompt

4: Compile a program

5: Present your work to the whole workshop assistant

6: Ask For more clarification, if any

Key readings 3.2.2 Compilation techniques

Key Compilation Techniques

- **Preprocessing**: Handles preprocessor directives and macros, performing text substitution to prepare the code for further processing.
- **Lexical Analysis (Scanning)**: Breaks down the modified source code into individual tokens, the smallest units in the programming language.
- **Syntax Analysis (Parsing)**: Checks the order and structure of tokens to ensure they follow syntax rules, constructing a parse tree or abstract syntax tree (AST).
- **Semantic Analysis**: Checks the meaning and correctness of the code, including type checking and enforcing complex semantic rules.
- **Intermediate Code Generation (Optional)**: Generates an intermediate representation of the code for optimizations, closer to machine code but platform-independent.
- **Optimization**: Improves code efficiency through techniques like constant folding, loop unrolling, and dead code elimination.
- **Code Generation**: Translates the code into machine code or assembly language specific to the target hardware platform.
- **Linking (if necessary)**: Resolves references between different parts of a program, combines them into a single executable, and links external libraries.
- **Output**: Produces an executable program or binary file that can be run on the target machine, containing the machine code the CPU can execute.

Points to Remember

What is a Compiler?

A **compiler** is software that converts high-level programming language code (like C) into machine-level language. It checks the code for errors and reports them if the code does not follow language rules.

Types of Compilers:

- **Borland Turbo C**: A basic, fast C compiler known for its small size, re-released as freeware in 2006.

- **Tiny C Compiler (TCC)**: Optimized for slow computers with limited disk space, known for its small size and fast compilation speed.
- **Portable C Compiler (PCC)**: An influential compiler from the mid-1970s, designed for portability and error checking with machine-dependent source files.
- **GCC (GNU Compiler Collection)**: Supports multiple languages, initially released for C, and known for extensive optimizations like dead code elimination.

Compilation techniques

- **Preprocessing**: Processes preprocessor directives and macros, performing text substitution to prepare the code for further processing.
- **Lexical Analysis (Scanning)**: Breaks down the source code into tokens, which are the smallest units of meaning in the programming language.
- **Syntax Analysis (Parsing)**: Analyzes the sequence of tokens to ensure they follow the syntax rules of the language, constructing a parse tree or abstract syntax tree (AST).
- **Semantic Analysis**: Verifies the meaning and correctness of the code, performing type checking and enforcing semantic rules.

Application of learning 3.2.

A parser is responsible for checking the syntactic structure of the code according to the rules of the programming language. Consider the following code snippet:

```
if (x > 0 {
   printf("Positive");
}
```

Task:

1. Identify the syntax error in the code and explain why it would be flagged by the parser.
2. How does a parser use a parse tree or abstract syntax tree (AST) to represent the structure of this code?

Indicative content 3.3: Test of the C program

Duration: 2 hrs

Theoretical Activity 3.3.1 Design of effective test cases

Tasks:

1: you are requested to perform tasks described below:
Write a c program to display a simple calculator and write all possible test cases of this program.
2: Discuss on the given tasks in their groups.
3: Write down the finding from the discussion.
4: Present the findings.
4: Ask clarification if any.
6: Read the key reading **3.3.1**

Key readings 3.3.1. Design of effective test cases

1. Testing in c programming

Understanding Requirements:

- **Functional Requirements:** Define what the program is supposed to do. For instance, if testing a function that calculates the factorial, ensure you understand how it should handle different inputs.
- **Non-Functional Requirements:** Consider performance, security, and usability. For example, test how the program handles large input sizes or potential buffer overflows.

Identify Key Features and Scenarios:

- **Feature Identification:** Determine the core features or functions of the program. For instance, if testing a sorting algorithm, key features include sorting accuracy and performance.
- **Scenario Development:** Create scenarios that cover all possible use cases, including edge cases. For a sorting function, scenarios might include empty lists, single-element lists, and large lists with various types of data.

Design Test Cases:

- **Test Case Components:**

✓ **Test Case ID:** Unique identifier for the test case.
✓ **Description:** Brief explanation of what the test case is checking.
✓ **Preconditions:** Conditions that must be met before executing the test case.

- ✓ **Test Steps:** Detailed steps to perform the test.
- ✓ **Expected Result:** The outcome that should occur if the test passes.
- ✓ **Postconditions:** State after the test has been executed.
- **Example Test Case for a Factorial Function:**
- ✓ **Test Case ID:** TC_FACTORIAL_01
- ✓ **Description:** Verify factorial calculation for positive integers.
- ✓ **Preconditions:** Function factorial(int n) is available.
- ✓ **Test Steps:**

1. Call factorial(5).
2. Compare the output with the expected result 120.

- ✓ **Expected Result:** Output should be 120.
- ✓ **Postconditions:** Function factorial(5) correctly returns 120.

Consider Edge Cases and Boundary Values:

- **Edge Cases:** Test the limits of the input domain. For instance, test the factorial function with large values to ensure it handles them without overflow.
- **Boundary Values:** Test the smallest and largest possible values for inputs, such as 0 or negative values in a function that is supposed to handle only non-negative integers.

Include Negative and Error Cases:

- **Negative Testing:** Ensure the program handles invalid inputs gracefully. For example, test a sorting function with unsorted input to check if it correctly sorts it.
- **Error Handling:** Verify that the program handles and reports errors appropriately. For instance, check if a function correctly handles division by zero.

Automate Where Possible:

- **Automation Benefits:** Automated tests can be run frequently and consistently, making it easier to catch regressions. Tools like CUnit or CMocka can help automate unit testing in C.
- **Test Scripts:** Write scripts to automate the execution of test cases and validation of results.

Document Test Cases Thoroughly:

- **Clear Documentation:** Ensure that each test case is well-documented to make it easy for others to understand and execute. Include details about the purpose of the test and how to interpret results.

Review and Refactor Test Cases:

- **Review:** Regularly review test cases to ensure they remain relevant as the code evolves. Update test cases based on new requirements or identified issues.
- **Refactor:** Refactor test cases to improve clarity and efficiency. Avoid redundancy and ensure that test cases are maintainable.

Perform Regression Testing:

- **Regression Testing:** After making changes to the code, run existing test cases to ensure that new changes haven't introduced new bugs.

Leverage Test Coverage Metrics:

- **Coverage Metrics:** Use tools to measure test coverage to ensure that all parts of the code are tested. Aim for high coverage to increase confidence in the code's correctness.

Understand the Requirements:

- **Functional Requirements:** Clearly understand what the software is supposed to do. Define what constitutes correct behavior for each function or feature.
- **Non-Functional Requirements:** Consider aspects like performance, security, and usability. For instance, check response times and handling of large inputs.

Define Testing Objectives:

- **Purpose:** Determine the objective of each test case. Are you validating a specific feature, checking for performance issues, or ensuring the software meets user requirements?
- **Scope:** Clearly define what each test case should cover, including specific inputs, processes, and outputs.

Identify Test Scenarios:

- **Feature Scenarios:** Develop scenarios based on different features or functionalities. Ensure you cover all critical paths and common use cases.
- **Edge Cases:** Include scenarios that test the boundaries of the input domain. For example, testing with minimum and maximum input values.

Design Test Cases:

- **Test Case Components:**

✓ **Test Case ID:** Assign a unique identifier for tracking.
✓ **Description:** Briefly describe what the test case is verifying.
✓ **Preconditions:** State any conditions that must be met before the test case can be executed.
✓ **Test Steps:** Outline the sequence of actions to be performed.
✓ **Expected Result:** Define the anticipated outcome for each test step.
✓ **Postconditions:** Describe the state after executing the test case.

- **Example:**

✓ **Test Case ID:** TC_LOGIN_01
✓ **Description:** Verify login functionality with valid credentials.
✓ **Preconditions:** User account with valid credentials exists.
✓ **Test Steps:**
Navigate to the login page.
Enter valid username and password.
Click the login button.
✓ **Expected Result:** User should be redirected to the dashboard.

Consider Boundary and Edge Cases:

- **Boundary Values:** Test at the limits of input ranges (e.g., the smallest and largest possible inputs).
- **Edge Cases:** Test unusual or extreme conditions that might not be frequently encountered but could cause issues.

Include Negative and Error Cases:

- **Invalid Inputs:** Test with incorrect or unexpected inputs to ensure the system handles errors gracefully (e.g., invalid user credentials).
- **Error Handling:** Verify that the system reports errors correctly and that the application does not crash.

Prioritize Test Cases:

- **Critical Tests:** Focus on the most crucial functionalities first, especially those that are critical to the application's core functionality.
- **High-Risk Areas:** Prioritize areas of the code that have been recently changed or have a history of defects.

Automate Where Possible:

- **Automation Benefits:** Automated tests can run frequently, help in regression testing, and provide consistent results.
- **Tools:** Utilize testing frameworks and tools that support automation for your development environment.

Review and Refactor Test Cases:

- **Review:** Regularly review test cases to ensure they are relevant and up-to-date with the latest application changes.
- **Refactor:** Optimize and refactor test cases to eliminate redundancy and improve clarity.

Document Test Cases:

- **Clear Documentation:** Ensure that test cases are well-documented to facilitate understanding and execution by different testers.
- **Maintainability:** Keep documentation organized and up-to-date with any changes in the application or testing requirements.

Leverage Test Coverage Metrics:

- **Coverage Analysis:** Use tools to measure test coverage and ensure that all critical parts of the application are tested.
- **Aim for High Coverage:** Strive for comprehensive test coverage to increase confidence in the software's reliability.

Conduct Peer Reviews:

- **Collaborative Review:** Have colleagues review test cases to catch issues that may have been missed and to get feedback on test design.

Execute and Monitor:

- **Execution:** Run the test cases and carefully monitor results.

- **Issue Tracking:** Log any defects or issues discovered during testing and track their resolution

Practical Activity 3.3.2: Debugging techniques

Task:

1: You are requested to perform tasks described below:
Write a c program to find the factorial of a number, you are requested to identify the debugging techniques you have used.
2: By using key readings **3.3.2**, Write the source code of a given scenario in your groups
3: Compile the written source code
4: Debug the founded bugs if any

Key readings 3.3.2 Debugging techniques

1. Debugging Techniques for C Programming:

- **Debugging** is an essential skill for C programmers, as it helps identify and correct errors in your code.

2. common debugging techniques and tools you can use when working with C programming:

- **Print Debugging:**

✓ Insert print statements in your code to display the values of variables and the program's flow. Printing variable values at different points in your code can help you identify where issues are occurring.

- **Debugger Tools:**

✓ Use a debugger tool, such as GDB (GNU Debugger) on Unix-like systems or integrated debugging features in IDEs like Visual Studio or Code::Blocks on Windows. Debuggers allow you to set breakpoints, step through code, inspect variables, and analyze program state during runtime.

- **Breakpoints:**

✓ Set breakpoints at specific lines in your code where you suspect issues may exist. When the program reaches a breakpoint, it pauses execution, allowing you to

inspect variables and step through the code.

- **Watch and Evaluate Expressions:**

✓ Many debuggers let you watch and evaluate expressions. You can monitor the values of specific variables or expressions in real-time as the program runs.

- **Stack Traces:**

✓ When an error occurs, examine the stack trace provided by the debugger. It shows the function call sequence and can help you trace the source of an issue.

- **Memory Debugging:**

✓ Use tools like Valgrind or AddressSanitizer to detect memory-related errors, such as memory leaks, buffer overflows, and invalid memory accesses.

- **Static Analysis Tools:**

✓ Employ static code analysis tools like Clang Static Analyzer, Lint, or Coverity to identify potential issues in your code without running it. These tools can catch issues at compile-time.

- **Assertions:**

✓ Insert assertions in your code to verify assumptions and conditions. If an assertion fails, it halts the program's execution, helping you identify issues early in development.

- **Code Review:**

✓ Have a colleague or peer review your code. A fresh set of eyes may spot issues that you missed.

- **Error Messages and Logs:**

✓ Check for error messages and logs generated by the program or external libraries. These messages can provide valuable information about what went wrong.

- **Unit Testing:**

✓ Write unit tests for your functions and modules. Unit tests are designed to test individual parts of your code in isolation, making it easier to identify and fix issues.

- **Regression Testing:**

✓ Use regression tests to ensure that new code changes do not introduce new

errors in existing functionality.

- **Isolate the Problem:**

✓ Isolate the problem to the smallest possible code snippet or module that reproduces the issue. Reducing the code's complexity can help pinpoint the problem more effectively.

- **Code Profiling:**

✓ Use profiling tools like gprof to identify performance bottlenecks in your code. Profiling helps optimize the program's performance.

- **Documentation and Comments:**

✓ Maintain clear documentation and comments in your code. Understanding the purpose and design of your code can make debugging more efficient.

- **Online Communities and Forums:**

✓ Participate in programming communities and forums like Stack Overflow to seek help and advice from experienced programmers when you're stuck.

Debugging is an iterative process, and it often requires a combination of these techniques to identify and resolve issues in your C programs effectively. The choice of technique may depend on the nature of the problem and your familiarity with the debugging tools and methods available.

Points to Remember

Design an effective test

- **Test Case ID:** Assign a unique identifier.
- **Description:** Briefly describe what the test case is verifying.
- **Preconditions:** State any conditions that must be met before executing the test case.
- **Test Steps:** Outline the sequence of actions to be performed.
- **Expected Result:** Define the anticipated outcome.
- **Postconditions:** Describe the state after the test has been executed

Debugging techniques

- **Print Debugging**: Use print statements to display variable values and program flow.
- **Debugger Tools**: Utilize tools like GDB or IDE features to set breakpoints, step through code, and inspect variables.

- **Breakpoints**: Pause execution at specific points to inspect variables and program state.
- **Watch and Evaluate Expressions**: Monitor and evaluate variable values or expressions during execution.

Application of learning 3.3.

you have opened a savings account at a local bank, and you want to know how much interest you will earn over a certain period. The bank offers a simple interest rate of 5% per year. You decide to write a C program to calculate the interest you will earn based on the amount of money (principal) you deposit, the interest rate, and the time period in years. For this task, please complete the following:

a. Identify possible test cases.

b. Describe the debugging techniques employed.

Learning outcome 3 end assessment

Theoretical assessment

Choose the letter corresponding to the correct answer:

1. **Which of the following is an example of a runtime error in C programming?**
   A) Using an undeclared variable
   B) Dividing a number by zero
   C) Missing a semicolon at the end of a statement
   D) Incorrect use of function syntax
2. **What type of error is caused by violations of the rules of the programming language syntax?**
   A) Runtime error
   B) Logical error
   C) Syntax error
   D) Semantic error
3. **Which type of error is the most difficult to detect and fix because the program still runs but gives incorrect output?**

A) Compilation error
   B) Logical error
   C) Syntax error
   D) Semantic error

4. **What type of error will the following code produce? printf("Hello World)**
   A) Syntax error
   B) Runtime error
   C) Logical error
   D) None of the above
5. **Which of the following is the first step in the C compilation process?**
   A) Linking
   B) Preprocessing
   C) Code Optimization
   D) Assembling
6. **What is the purpose of the linker in the compilation process?**
   A) To check syntax errors
   B) To combine multiple object files into a single executable
   C) To translate code into machine language
   D) To optimize the code for performance

7. **Which compilation technique translates high-level code into machine code one line at a time, while the program is running?**
    A) Interpreter
    B) Compiler
    C) Assembler
    D) Linker
8. **Which phase of compilation involves translating assembly code to machine code?**
    A) Linking
    B) Preprocessing
    C) Assembling
    D) Optimization
9. **Which of the following tools is commonly used for debugging C programs on Unix-like systems?**
    A) Valgrind
    B) Git
    C) NetBeans
    D) Makefile
10. **What does a debugger tool primarily help with?**
    A) Writing code faster
    B) Detecting syntax errors
    C) Testing program performance
    D) Finding and fixing runtime errors
11. **Which debugging tool allows you to step through your program line-by-line to check its execution flow?**
    A) Compiler
    B) Debugger
    C) Profiler
    D) Assembler
12. **Which of the following is NOT a feature of most modern debugging tools?**
    A) Breakpoints
    B) Memory leak detection
    C) Automated code generation
    D) Stack trace analysis
13. **Which tool is used to analyze memory management problems, like memory leaks and invalid memory access in C programs?**
    A) GDB (GNU Debugger)
    B) GCC (GNU Compiler Collection)
    C) Valgrind
    D) Make

14. What is the main use of setting breakpoints while debugging a program?

A) To make the code run faster

B) To stop the execution of the program at specific points

C) To compile the program

D) To link multiple object files

Subjective questions

1. What is the purpose of a compiler in the context of C programming?
2. Describe the main phases of the compilation process in C programming.
3. What is the role of a linker in the C compilation process?
4. Explain the difference between syntax errors and semantic errors in C programming.
5. What is the purpose of the #include directive in a C program?
6. How does optimization improve the performance of a compiled C program?
7. What is the significance of the main function in a C program?
8. What is a preprocessor directive, and provide an example?

Practical assessment

1. **Write a C program, which calculates the area of a rectangle.**

Your tasks are:

1. Compile the C program using a C compiler.
2. Identify and explain any errors or warnings that occur during the compilation process.
3. Describe the steps taken during the compilation and the purpose of each step.

References

T. F. L. Y. Presents, “Practical C Programming , 3rd Edition By Steve Oualline 3rd Edition August 1997 ISBN : Table of Contents Getting Help in an Integrated Development Environment,” no. August, pp. 1–70, 1997.

S. G. Kochan, *Programming in C Warning and Disclaimer Bulk Sales*. 2004.

M. S. Ummah, “No 主観的健康感を中心とした在宅高齢者における 健康関連指標に関する共分散構造分析 Title,” *Sustain.*, vol. 11, no. 1, pp. 1–14, 2019, [Online]. Available: http://scioteca.caf.com/bitstream/handle/123456789/1091/RED2017-Eng-8ene.pdf?sequence=12&isAllowed=y%0Ahttp://dx.doi.org/10.1016/j.regsciurbeco.2008.06.005%0Ahttps://www.researchgate.net/publication/305320484_SISTEM_PEMBETUNGAN_TERPUSAT_STRATEGI_MELESTARI

S. C. Dewhurst and K. Stark, *Programming in C++*, vol. 2, no. 4. 1991. doi: 10.1145/126983.126989.

M. Vine, *C Programming for the Absolute Beginner*. 2008.

https://www.bing.com/ck/a?!&&p=be8bdb80a0c6fbf0JmltdHM9MTcwMDM1MjAwMCZpZ3VpZD0wMTU4N2M4NC03NWVlLTZlOWItMzIwYS02ZjQ0NzQ2YzZmM2EmaW5zaWQ9NTE5OA&ptn=3&ver=2&hsh=3&fclid=01587c84-75ee-6e9b-320a-6f44746c6f3a&psq=scenario+application+of+learning+of+debugging+c+programming&u=a1aHR0cHM6Ly9jcy5icm93bi5lZHUvY291cnNlcy9jc2NpMTMxMC8yMDIwL2Fzc2lnbi9sYWJzL2xhYjEuaHRtbA&ntb=1

https://www.educba.com/best-c-compilers/

https://www.google.com/search?client=firefox-b-d&q=common+debugging+techniques+and+tools+you+can+use+when+working+with+C+programming

https://www.google.com/search?client=firefox-b-d&q=compiler+in+c+programming

https://www.quora.com/What-are-the-different-types-of-compilers-in-C

RTB | RWANDA TVET BOARD

October, 2024